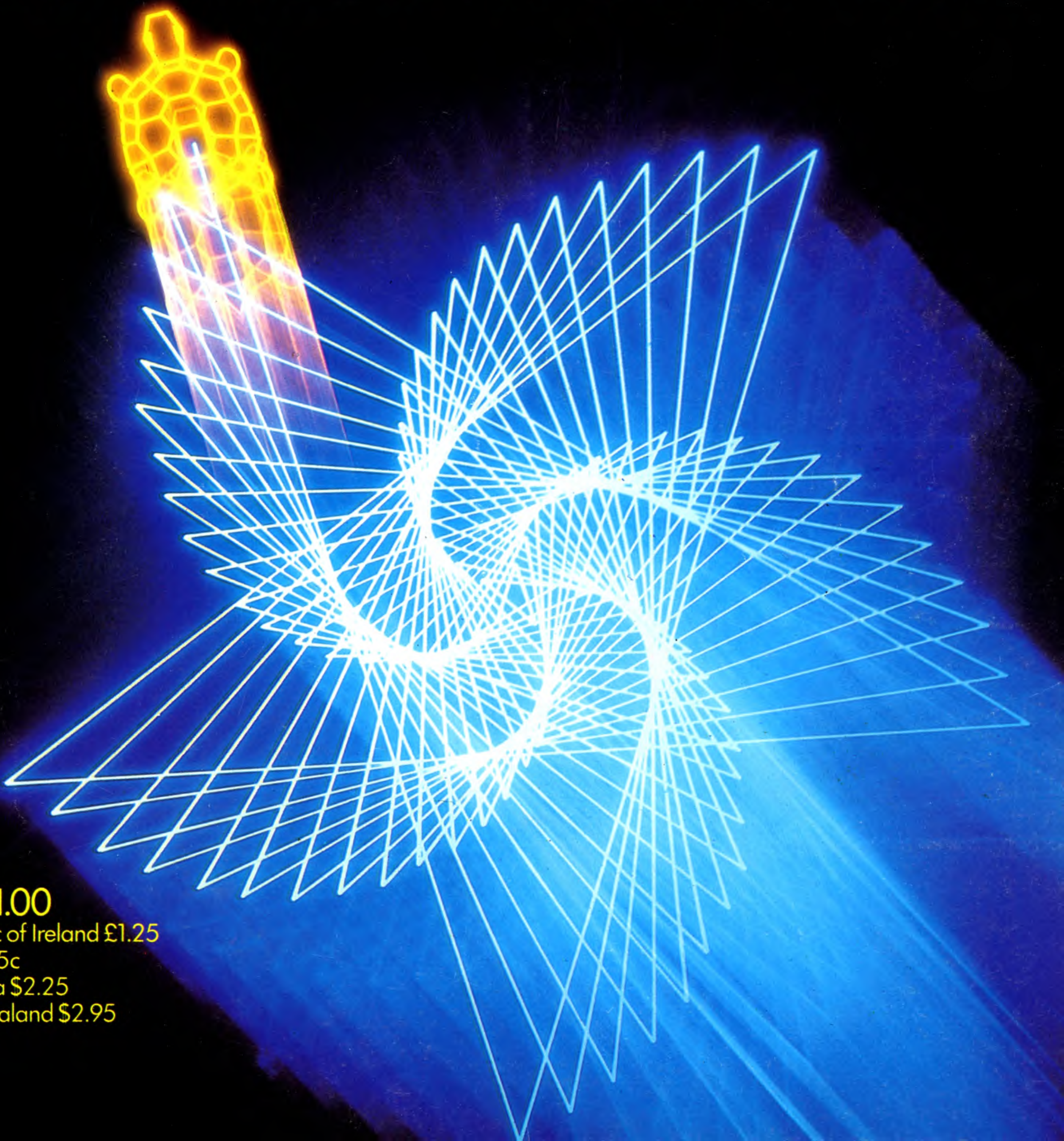
A MARSHALL CAVENDISH 41 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95

INPUT

Vol. 4 No 41

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS

Front cover, Paul Chave. Pages 1269, 1270, 1271, 1272, 1273, 1274, 1275, Joe Wright. Page 1276, Simon McBride/Chris Lyon. Page 1278, Ray Duns/Chris Lyon. Page 1283, Dave King. Pages 1284, 1285, George Logan. Pages 1289, 1290, 1291, 1292, 1293, 1295, Kevin O'Keefe. Pages 1296, 1298, 1299, 1300, Paul Chave.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Typeset by MS Filmsetting Limited, Frome, Somerset. Printed by Cooper Clegg Web Offset Ltd, Gloucester and Howard Hunt Litho, London.



There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland:
Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd, Department 980, Newtown Road, Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services, Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:

Subscription Department, Marshall Cavendish Services Ltd, Newtown Road, Hove, Sussex BN3 7DN

Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

A COMPUTER INTERIOR DESIGNER

- THE STRENUOUS METHOD
- USING GRAPH PAPER
- MEASURING YOUR ROOM
- SETTING A SCALE
- ROTATING YOUR FURNITURE

Let your home computer take the strain out of planning the layout of any room so you can move your furniture or refit your kitchen simply by pressing a few keys

Planning a room in order to get the optimum layout for all the various items of furniture can be a tricky problem. Most people tackle it in one of two different ways.

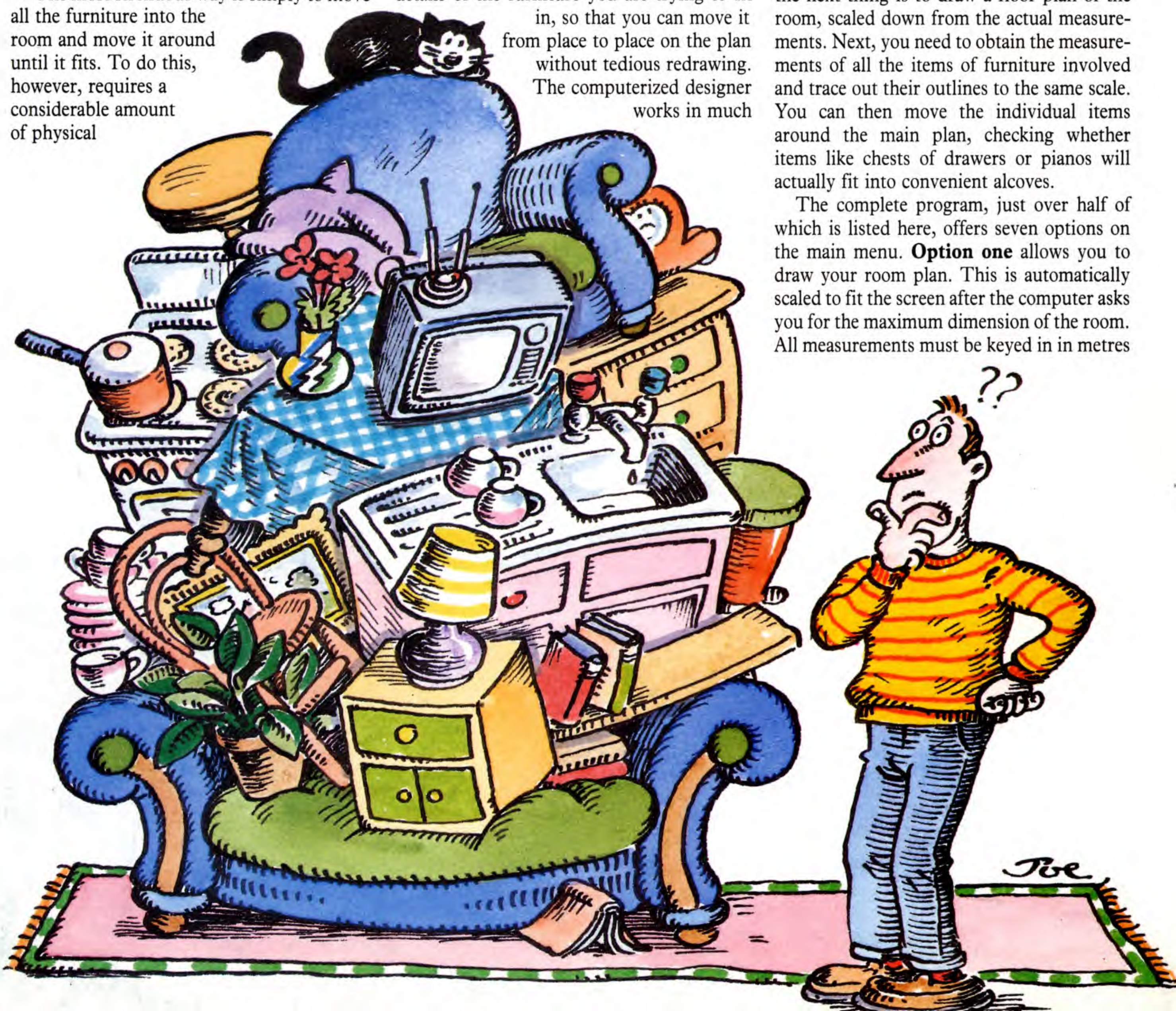
The most strenuous way is simply to move all the furniture into the room and move it around until it fits. To do this, however, requires a considerable amount of physical effort, and there is no guarantee it will all go into the space available.

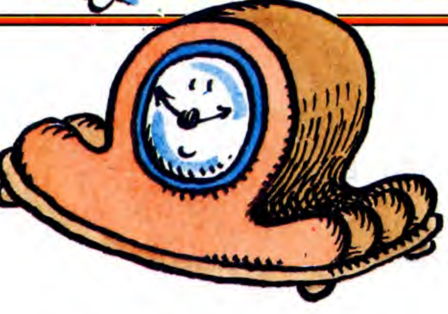
The simpler method is to draw an accurate scale plan, and work it all out on paper first. It's a lot easier to draw a new rectangle on your plan than to move a fridge!

Now, there's a third alternative, using your home micro. Its graphic display can be used just like pencil and paper, with the advantage that it is easier to make corrections. And its memory can be used to store details of the furniture you are trying to fit in, so that you can move it from place to place on the plan without tedious redrawing. The computerized designer works in much the same way as you would draw a paper plan. The first thing to do is to measure the room accurately, preferably using a proper metal measuring tape. (A piece of string and a short ruler can do the job, but are likely to give you a cumulative inaccurate measurement and when planning a kitchen a couple of centimetres can make a big difference.)

Once you have all the measurements, including areas like alcoves around chimney breasts and the positions of windows or doors, the next thing is to draw a floor plan of the room, scaled down from the actual measurements. Next, you need to obtain the measurements of all the items of furniture involved and trace out their outlines to the same scale. You can then move the individual items around the main plan, checking whether items like chests of drawers or pianos will actually fit into convenient alcoves.

The complete program, just over half of which is listed here, offers seven options on the main menu. **Option one** allows you to draw your room plan. This is automatically scaled to fit the screen after the computer asks you for the maximum dimension of the room. All measurements must be keyed in in metres





and given a direction (Up, Down, Left or Right). When drawing the sections of wall, the computer offers you two options of direction and distance on each section which allow you to draw diagonals. You can also insert doors and windows where necessary.

Once you have drawn the room plan, **option two** allows you to position certain standard-sized pieces of furniture which are already defined in the program. All these items relate to the kitchen—possibly the most difficult room in the house to plan—and include a cupboard, cooker, washing machine, sink and fridge. Only one cupboard, for example, is necessary as any of these items can be inserted as many times as you wish. A key press allows you to select an item, another allows you to position it. And there's also a facility which allows you to rotate the item to the angle you require.

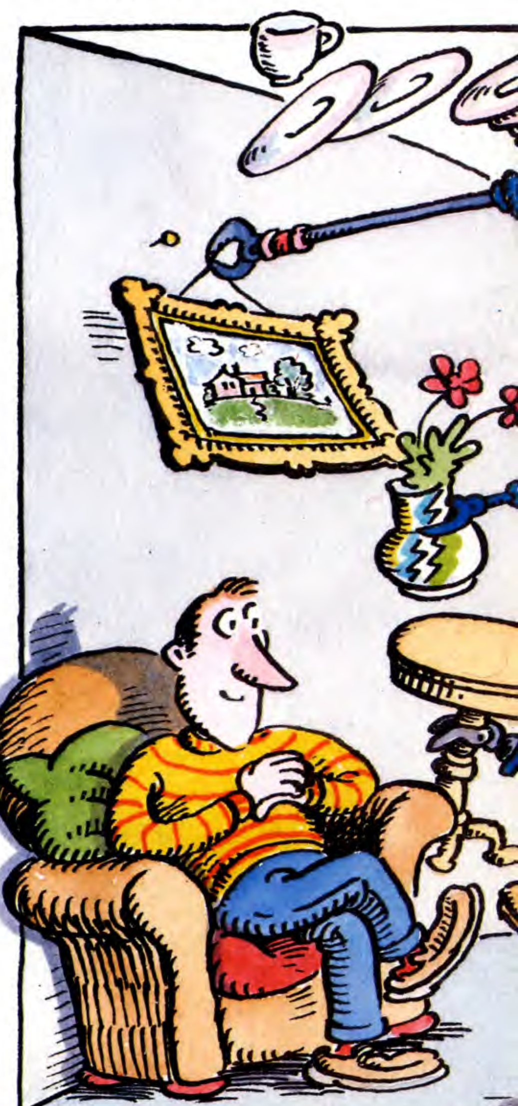
Option three allows you to redefine the size and shape of the five objects already contained within the program and define five more. Each shape can have no more than eight sides (except on the Dragon), but this is more than enough for most items of furniture. The new furniture is also automatically scaled to the length of the room you enter.

Option four allows you to save your design (option one) and its contents (options two and three) to tape or disk. Make sure you have plenty of space on tape.

With **option five**, you can load a previously saved design. This is mainly for use with **option six**—the print option. Only the Spectrum can print out the screen display directly, so this option is not complete for the other computers. However, the routine can be used to call either a machine code screen dump or separate BASIC program to do this. A future article in *INPUT* will give suitable screen dumps, which will enable you to produce a hard copy of your design. **Option seven** allows you to quit the program.

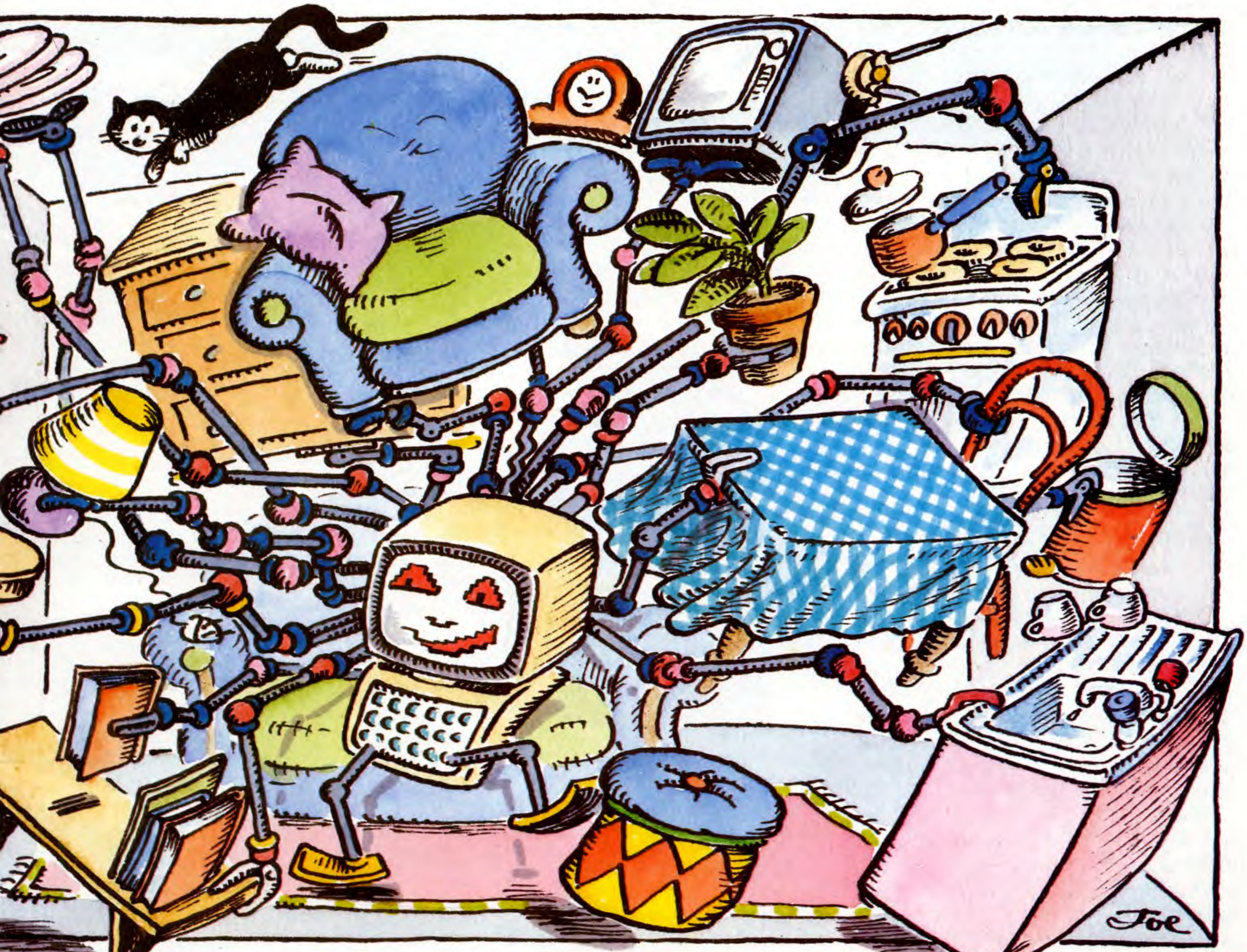
The second part of this article tells you how to use the different options and lists the final part of the program. You will need to key in the entire program before any of the options can be used, so if you enter this part separately, SAVE it to tape without running it.

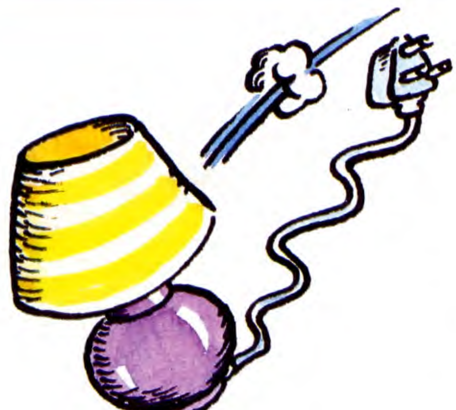
```
5 POKE 23658,8
10 BORDER 0: PAPER 0: INK 4: CLS
12 GOSUB 8000
20 GOSUB 7000: PRINT INVERSE 1;
   AT 2,23;"[1]ROOM";AT 3,23;"[2]PLAN";
   AT 4,23;"[3]EQUIP";AT 5,23;"[4]SAVE";
   AT 6,23;"[5]LOAD";AT 7,23;"[6]PRINT";
   AT 8,23;"[7]QUIT"
30 LET K$="1234567": GOSUB 7040:
   GOSUB 7000: IF Z=55 THEN STOP
40 GOSUB 1000*(Z-49)+1000: GOTO 20
1000 LET NF=1: GOSUB 6080
1005 PLOT 0,0: LET X=0: LET Y=0
1010 PRINT PAPER 2; INK 6;AT 21,
   22;"MAXIMUM=";MAX
1015 PRINT INVERSE 1;AT
   2,22;"[W]WINDOW ";AT
   3,22;"[D]DOOR ";AT 4,22;"[B]BK
   WALL";AT 5,22;"[□]WT WALL";AT
   6,22;"[Q]QUIT ";AT 7,22;" "
1016 LET K$="WDBQ□": GOSUB 7040:
   INK 3*(Z=87)+2*(Z=68)+7*(Z=32)
1020 IF Z=81 THEN INK 4: RETURN
1025 GOSUB 7010: LET DX=X+D*
   ((D$="R")*SC)-D*((D$="L")*SC):
   LET DY=Y+D*((D$="U")*SC)-D*
   ((D$="D")*SC): GOSUB 7010
1030 LET DX=DX+D*((D$="R")*SC)-D*
   ((D$="L")*SC): LET DY=DY+D
   *((D$="U")*SC)-D*((D$="D")*SC)
1032 IF DX>175 OR DX<0 OR DY>175
   OR DY<0 THEN PRINT FLASH 1;AT
   7,23;" ERROR ": PAUSE 100: GOTO 1015
1035 DRAW DX-X,DY-Y: LET X=PEEK
   23677: LET Y=PEEK 23678: LET
   X$=STR$ (X/SC): LET Y$=STR$ (Y/SC)
1037 IF LEN X$<3 THEN LET X$=F$( TO
   3-LEN X$)+X$
1038 IF LEN Y$<3 THEN LET Y$=F$( TO
   3-LEN Y$)+Y$
1039 LET X$=X$( TO 3): LET Y$=Y$( TO
   3): PRINT INK 7;AT 10,24;"DX=";X$;AT
   11,24; "DY=";Y$
1040 GOTO 1016
2000 IF NF=0 THEN RETURN
2005 FOR N=1 TO 10: LET Q$="["+STR$
   N+"]"+O$(N): PRINT INVERSE 1;AT
   N*2,24+4-LEN Q$;Q$: NEXT N
2010 LET R=0: GOSUB 6060: LET CU=2
2020 PRINT INVERSE 1;AT CU,29;CHR$ 144:
   FOR N=1 TO 10: NEXT N: PRINT
   INVERSE 1;AT CU,29;"□"
2030 LET K$=INKEY$: LET CU=CU-
   2*(((K$="7")*(CU>2)))+2*(((K$=
   "6")*(CU<20)))
2040 IF K$<>"S" THEN GOTO 2020
2050 LET OB=CU/2: LET OX=85: LET
   OY=85
2057 LET F=1: GOSUB 6010
2060 GOSUB 7000: PRINT AT 5,24; FLASH 1;
   "OBJ=";O$(OB); FLASH 0;AT 7,22;
   INVERSE 1;"[5-8] MOVE";AT 8,22;"[P]
   PLACE";AT 9,22;"[C] CLKWSE";AT 10,22;
   "[A] ANTCLK";AT 11,22;"[Q] QUIT"
2070 LET K$="5678PACQ": GOSUB 7040:
   IF Z=81 THEN LET R=0: LET F=1:
   GOSUB 6010: GOTO 6060
2080 LET F=1: GOSUB 6010: LET
   OX=OX+2*((Z=56)*(OX<175))-
   2*((Z=53)*(OX>1)): LET OY=OY+2*
   ((Z=55)*(OY<175))-2*((Z=54)*
   (OY>0))
2085 IF Z=67 OR Z=65 THEN GOSUB 6020
2090 GOSUB 6010
2100 IF Z=80 THEN LET F=0: GOSUB
   6010: GOSUB 7000: GOTO 2000
2110 GOTO 2070
3000 PRINT AT 2,24;"DESIGN"
3012 INPUT "ENTER FIGURE TO
   REDEFINE ";OB: IF OB<1 OR OB>10
   THEN GOTO 3012
```



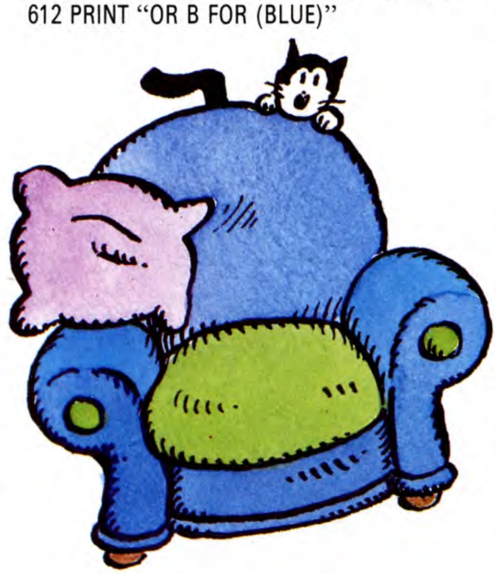
```
3013 INPUT "ENTER NUMBER OF SIDES
   (1-15) ?";S(OB): IF S(OB)<1 OR
   S(OB)>15 THEN GOTO 3013
3014 INPUT "TWO LETTER IDENTITY
   CODE? ";O$(OB)
3016 LET R=0: GOSUB 6060: LET OX=80:
   LET OY=80
3017 FOR S=1 TO S(OB)*2 STEP 2
3020 FOR N=1 TO 2: GOSUB 7010: LET
   D=D/125: LET O(OB,S)=O(OB,S)+((D*
   (-1*(D$="L")+(D$="R")))): LET
   O(OB,S+1)=O(OB,S+1)+((D*(-1*
   (D$="D")+(D$="U"))))
3030 NEXT N
3040 NEXT S
3050 LET F=1: GOSUB 6010: INPUT "IS
   THIS CORRECT (Y OR N)? ";S$: IF
   S$="N" THEN GOSUB 6010: FOR N=1
   TO S(OB)*2+1: LET O(OB,N)=0: NEXT
   N: GOTO 3012
3060 GOSUB 6010: RETURN
4000 GOSUB 6200
4015 IF Z=83 THEN
   SAVE E$SCREEN$ :
   RETURN
4020 SAVE E$ DATA O()
4030 SAVE E$ DATA S()
4040 SAVE E$ DATA O$()
4050 RETURN
5000 GOSUB 6200
5010 IF Z=83 THEN LOAD E$SCREEN$ :
   RETURN
5020 LOAD E$ DATA O()
5030 LOAD E$ DATA S()
5040 LOAD E$ DATA O$()
5050 RETURN
6000 COPY : RETURN
6010 OVER F: INK 7: PLOT OX,OY: FOR N=1
   TO (S(OB)*2) STEP 2:
   DRAW O(OB,N)*CY-O(OB,N+1)*CX,
   O(OB,N)*CX+O(OB,N+1)*CY: NEXT N
6014 OVER 0: INK 4: RETURN
6020 LET R=R+(2*(Z=65))-
   (2*(Z=67)): IF R>360
   THEN LET R=0
6030 IF R<0 THEN LET R=360-R
6060 LET A=R*(PI/180): LET CY=SC*COS
   A: LET CX=SC*SIN A: RETURN
6080 INPUT "ENTER MAXIMUM
   DIMENSION? ";MAX
6090 LET SC=175/MAX
6100 RETURN
6200 PRINT INVERSE 1;AT 10,23;
   "[S]SCREEN";AT 11,23;"[V]VAR'S ": LET
   K$="SV": GOSUB 7040: INPUT "ENTER
   FILENAME ";E$: RETURN
7000 FOR N=0 TO 21: PRINT PAPER 4;AT
   N,22;"□□□□□□□□□□": 
   NEXT N: PRINT AT 0,25;"MENU ":
   RETURN
```

```
0 IF FG=1 THEN GOSUB 30010:
   CSET(2):GOTO 270
1 POKE 51,255:POKE 52,94:POKE 55,255:POKE
   56,94:CLR
```





```
2 FOR Z=2432Ø TO 24431:READ X:
   POKE Z,X:NEXT Z:FG=1
3 DATA 169,Ø,141,14,22Ø,169,53,133,1
4 DATA 169,Ø,133,251,133,253,169,224,
   133,252,169,96,133,254,16Ø,Ø
5 DATA 177,251,145,253,192,63,2Ø8,16,
   165,252,2Ø1,255,2Ø8,1Ø
6 DATA 162,1,142,14,22Ø,162,55,134,1,
   96,2ØØ
7 DATA 2Ø8,229,23Ø,252,23Ø,254,76,25,
   95
8 DATA 165,45,133,253,165,46,133,254
9 DATA 162,8,16Ø,1,169,1,32,186,255,
   162,113,16Ø,95,173,112,95
1Ø DATA 32,189,255,169,Ø,133,251,169,
   96,133,252,162,Ø,16Ø,128,169,251
11 DATA 32,216,255,165,253,133,45,165,
   254,133,46,96
2Ø HIRES 3,6:COLOUR 13,6
25 PRINT "◘▓":PT=1:GOSUB 7Ø
3Ø GOTO 27Ø
7Ø DIM DE(2Ø,9)
8Ø FOR F=1 TO 1Ø:H$(F)="□□":
   NEXT F
11Ø FOR P=1 TO 9 STEP 2
12Ø FOR T=1 TO 9
13Ø READ DE(P,T),DE(P+1,T)
14Ø NEXT T,P
16Ø DATA Ø.3,Ø.25,−Ø.6,Ø,Ø,−Ø.5,Ø.6,Ø,Ø,
   Ø.5,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
17Ø DATA Ø.255,Ø.255,−Ø.55,Ø,Ø,−Ø.55,
   Ø.55,Ø,Ø,Ø.55,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
18Ø DATA Ø.3,Ø.25,−Ø.6,Ø,Ø,−Ø.5,Ø.6,Ø,Ø,
   Ø.5,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
19Ø DATA Ø.5,Ø.25,−1,Ø,Ø,−Ø.5,1,Ø,Ø,Ø.5,
   Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
2ØØ DATA Ø.255,Ø.255,−Ø.55,Ø,Ø,−Ø.55,
   Ø.55,Ø,Ø,Ø.55,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
21Ø FOR T=1 TO 5:READ H$(T):
   NEXT T
22Ø DATA CU,CO,WA,SI,FR
23Ø I=Ø:J=Ø:FL=Ø:F1=Ø:
   F2=Ø:F3=Ø:SC=5Ø
24Ø RETURN
27Ø LOW COL 5,6,6:TEXT 5,16Ø,"1)□
   PLAN□ROOM□□□□□□□□□
   □□2)□DESIGN□LAYOUT",1,1,8
29Ø TEXT 5,168,"3)□DESIGN□
   EQUIPMENT□□□4)□SAVE
   DESIGN",1,1,8
3ØØ TEXT 5,176,"5)□LOAD□
   DESIGN□□□□□□□□□6)
   □PRINT□□7)END",1,1,8:LOW COL
   4,6,Ø
31Ø TEXT Ø,188,"PRESS NUMBER OF
   YOUR CHOICE",1,1,12
34Ø POKE 198,Ø:HI COL
36Ø GET N$:IF N$<"1" OR N$>"7" THEN
   36Ø
37Ø N=VAL(N$)
38Ø ON N GOSUB5ØØ,96Ø,173Ø,21ØØ,219Ø,
   229Ø,49Ø:GOTO 27Ø
49Ø PRINT "◘▤":NRM:END
5ØØ OX=58:OY=159
52Ø HIRES 3,6
53Ø CSET(Ø):PRINT "◘▣ENTER MAXIMUM
   LENGTH OF ROOM (METRES)"
531 INPUT LR:IF LR<1 THEN 531
54Ø L$="LENGTH"+STR$(LR)+
   "□METRES":TEXT 82,Ø,L$,1,1,8
55Ø SC=151/LR
6ØØ IF OX<Ø THEN OX=Ø
6Ø1 IF OY<Ø THEN OY=Ø
6Ø2 IF OY>319 THEN OY=319
6Ø3 IF OY>167 THEN OY=167
6Ø9 PLOT OX,OY,1
61Ø CSET(Ø):PRINT "◘▣PRESS W FOR
   WINDOW":PRINT"OR D FOR DOOR"
611 PRINT "OR (SPACE) FOR WALL (CYAN)"
612 PRINT "OR B FOR (BLUE)"
```



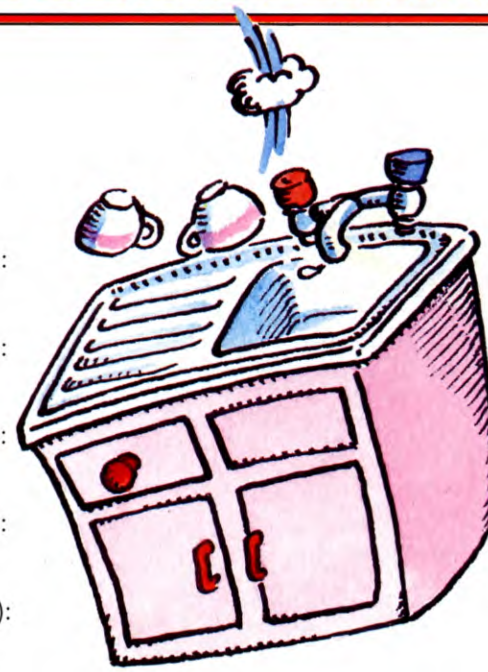
To use this program with Hi-Res delete Lines 1–11 and insert

1 FG=1

Replace all SYS calls with colons.

```
613 GET Q$
614 IF Q$="C" OR Q$="D" THEN FL=1
62Ø IF Q$="□" THEN PT=1
63Ø IF Q$="B" THEN PT=Ø
64Ø IF Q$<>"W" AND Q$<>"D" AND
   Q$<>"□" AND Q$<>"B" THEN
   613
66Ø CSET(2):GOSUB 78Ø
661 IF Q$="D" OR Q$="W" THEN 672
663 GX=OX+SC*(NX(1)+NX(2)):
   GY=OY+SC*(NY(1)+NY(2))
664 IF GX<Ø OR GX>319 OR GY<Ø OR
   GY>159 THEN 68Ø
67Ø LINE OX,OY,GX,GY,PT
671 GOTO 68Ø
672 EX=OX+SC*(NX(1)+NX(2)):
   EY=OY+SC*(NY(1)+NY(2))
673 DX=EX−OX:DY=EY−OY:
   SX=SGN(DX):SY=SGN(DY):
   X2=SX:Y2=Ø:M=ABS(DX):
   N=ABS(DY)
674 IF M<=N THEN X2=Ø:Y2=SGN
   (DY):M=ABS(DY):N=ABS(DX)
675 S=INT(M/2):SH=S:FORI=1TOS:IF
   OX<ØOROX>319OROY<ØOROY>159
   THENI=M:GOTO 679
676 PLOT OX,OY,1:S=S+N:IF S<SH THEN
   678
677 S=S−M:OX=OX+SX*2:
   OY=OY+SY*2:GOTO 679
678 OX=OX+X2*2:OY=OY+Y2*2
679 NEXT I:GOTO 69Ø
68Ø OX=OX+SC*(NX(1)+NX(2)):
   OY=OY+SC*(NY(1)+NY(2))
69Ø GOSUB 3ØØ2Ø:CSET(Ø):PRINT
   "◘PRESS C TO CLEAR ROOM PLAN"
692 PRINT "OR F WHEN FINISHED PLAN"
693 PRINT "OR <SPACE> TO CONTINUE"
7ØØ GET C$:IF C$="" OR (C$<>"C"
   AND C$<>"□" AND C$<>"F")
   THEN 7ØØ
7Ø2 IF C$="C" THEN HIRES 13,6:
   OX=58:OY=159
71Ø IF C$="□" THEN F1=Ø
72Ø IF C$="F" THEN F1=1:CSET(2)
73Ø IF F1=Ø THEN 6ØØ
76Ø GOSUB 27Ø
77Ø RETURN
78Ø CO=1
82Ø BLOCKØ,168,319,199,Ø
825 TEXTØ,168,"ENTER DIRECTION (U,D,L,R)
   SPACE−Ø",1,1,8
83Ø GETA$:IFA$=""OR(A$<>"U"AND
   A$<>"D"ANDA$<>"L"ANDA$<>
   "R"ANDA$<>"□")THEN 83Ø
831 TEXT 3ØØ,168,A$,1,1,8:
   DI$(CO)=A$
832 IF A$<>"□" THEN TEXT Ø,176,
   "ENTER DISTANCE",1,1,8:
   GOSUB1ØØØØ
833 DI(CO)=VAL(C$)
```

```
850 BLOCK Ø,168,319,199,Ø
870 CO=CO+1:IF CO<3 THEN 82Ø
880 FOR I=1 TO 2
890 IF DI$(I)="U" THEN NY(I)=
   -DI(I):NX(I)=Ø
900 IF DI$(I)="D" THEN NY(I)=
   DI(I):NX(I)=Ø
910 IF DI$(I)="L" THEN NX(I)=
   -DI(I):NY(I)=Ø
920 IF DI$(I)="R" THEN NX(I)=
   DI(I):NY(I)=Ø
930 IF DI$(I)="□" THEN NX(I)=
   Ø:NY(I)=Ø
940 NEXT I
950 RETURN
960 LOW COL 1,6,Ø
970 BLOCK Ø,16Ø,319,199,Ø
980 PH=Ø:C=SC*COS(PH):
   S=SC*SIN(PH)
1020 FOR F=1 TO 5:TEXT 67+((F-1)*5Ø),
   171,STR$(F)+"□"+H$(F),1,1,8
1030 TEXT 67+((F-1)*5Ø),187,STR$
   (F+5)+"□"+H$(F+5),1,1,8:NEXT F:
   HI COL
1040 GOSUB 131Ø
1050 GOSUB 1Ø7Ø
1060 RETURN
1070 TA(1)=35:TA(2)=193:I=1:G=1
1100 P1=TA(1):P2=TA(2):GOSUB 127Ø
1130 CE=1:PT=1:P1=TA(1):P2=TA(2):
   GOSUB 127Ø
1140 IF PG=1 THEN P3=I*2-1:RF=1:
   PT=2:GOSUB147Ø:GOSUB147Ø:RF=Ø
1145 CP=PEEK(197):RF=Ø
1150 IF CP=12 AND TA(1)-16>Ø THEN
   TA(1)=TA(1)-16
1155 IF CP=23 AND TA(1)+16<319 THEN
   TA(1)=TA(1)+16
1160 IF CP=5Ø AND TA(2)-16>Ø THEN
   TA(2)=TA(2)-16
1170 IF CP=55 AND TA(2)+16<199 THEN
   TA(2)=TA(2)+16
1180 IF CP=13 THEN GOSUB 131Ø
1190 IF CP=41 THEN GOSUB 138Ø
1200 IF CP=1Ø THEN GOSUB 153Ø:GOTO
   114Ø
1210 P1=TA(1):P2=TA(2):GOSUB 127Ø
1213 IF PEEK(197)=49 THEN PG=PG+1:IF
   PG>1 THEN PG=Ø:WAIT 197,64
1220 IF CP<>62 THEN 113Ø
1230 P1=TA(1):P2=TA(2):GOSUB 127Ø
1250 BLOCK Ø,16Ø,319,199,Ø:GOSUB
   27Ø:RETURN
1270 TEXT P1,P2,"*",2,1,8:RETURN
1310 BLOCK 8,192,24,198,Ø:I=1+INT
   ((TA(1)-75)/5Ø):IF TA(2)>183 THEN
   I=I+5
1315 IF I<1 THEN I=1
1370 TEXT Ø,192,STR$(I),1,1,8:RETURN
1380 P1=TA(1):P2=TA(2):P3=I*2-1:
   P$=H$((I+1)/2):GG=I:GOSUB 147Ø:
   RETURN
1470 G=P3
1485 EX=P1:EY=P2
1490 EX=EX+DE(G,1)*C+ DE(G+1,1):
   EY= EY+ DE(G,1)* S+ DE (G+1,1):
   NB=2
1491 HB=EX:LB=EY:GOSUB1521
1492 EX=EX+(DE(G,2)*C+DE(G+1,2)*S):
   EY=EY-(DE(G,2)*S-DE(G+1,2)*C):
   NB=3:GOSUB1521
1494 EX=EX+(DE(G,3)*C+DE(G+1,3)*S):
   EY=EY-(DE(G,3)*S-DE(G+1,3)*C):
   NB=4:GOSUB1521
1496 EX=EX+(DE(G,4)*C+DE(G+1,4)*S):
   EY=EY-(DE(G,4)*S-DE(G+1,4)*C):
   NB=5:GOSUB1521
1498 EX=EX+(DE(G,5)*C+DE(G+1,5)*S):
   EY=EY-(DE(G,5)*S-DE(G+1,5)*C):
   NB=6:GOSUB1521
1500 EX=EX+(DE(G,6)*C+DE(G+1,6)*S):
   EY=EY-(DE(G,6)*S-DE(G+1,6)*C):
   NB=7:GOSUB1521
1502 EX=EX+(DE(G,7)*C+DE(G+1,7)*S):
   EY=EY-(DE(G,7)*S-DE(G+1,7)*C):
   NB=8:GOSUB1521
1504 EX=EX+(DE(G,8)*C+DE(G+1,8)*S):
   EY=EY-(DE(G,8)*S-DE(G+1,8)*C):
   NB=9:GOSUB1521
1506 EX=EX+(DE(G,9)*C+DE(G+1,9)*S):
   EY=EY-(DE(G,9)*S-DE(G+1,9)*C)
1510 IF RF=Ø AND CE<>ØTHEN TEXT
   HB+1,LB+1,H$(GG),1,1,8:RETURN
1515 IF RF=Ø AND CE=ØTHEN TEXT
   HB+1,LB+1,H$(GG),Ø,1,8
1520 RETURN
1521 IF EX<Ø OR EX>319 OR EY<Ø OR
   EY>199 THEN RETURN
1522 FX=EX+(DE(G,NB)*C+
   DE(G+1,NB)*S):FY=EY-
   (DE(G,NB)*S-DE(G+1,NB)*C)
1523 IF FX<Ø OR FX>319 OR FY<Ø OR
   FY>199 THEN RETURN
1524 LINE EX,EY,FX,FY,PT
1525 IF EX<HB AND EY<LB THEN
   HB=EX:LB=EY
1526 RETURN
1530 RF=1:GG=I:CE=1
1555 J=I*2-1:AX=TA(1):
   AY=TA(2)
1570 C=SC*COS(PH):
   S=SC*SIN(PH)
1590 PT=2:P1=AX:P2=AY:P3=J:P$=H$
   ((J+1)/2):GOSUB 147Ø
1600 CP=PEEK(197):IF CP=64
   THEN 16ØØ
1601 PZ=AX:P2=AY:PT=2:P3=J:
   GOSUB 147Ø
1605 IF CP=12 AND AX-2>Ø THEN
   AX=AX-2
1610 IF CP=23 AND AX+2<319 THEN
   AX=AX+2
```



```
1620 IF CP=5Ø AND AY-2>Ø THEN
   AY=AY-2
1630 IF CP=55 AND AY+2<199 THEN
   AY=AY+2
1640 IF CP=37 THEN PH=PH+.Ø873
1650 IF CP=42 THEN PH=PH-.Ø873
1660 IF CP=14 THEN CE=Ø:GOTO 1686
1670 C=SC*COS(PH):S=SC*SIN(PH)
1675 PT=2:P1=AX:P2=AY:P3=J:IF
   CP=62 THEN 1686
1680 GOSUB 147Ø
1681 IF CP=41 THEN 1686
1685 GOTO 16ØØ
1686 RF=Ø:PT=CE:P1=AX:P2=AY:P3=J:
   IF CP=41 OR CP=14 THEN
   GOSUB 147Ø
1710 C=SC*COS(PH):S=SC*SIN(PH):IF
   CE=Ø THEN CE=1:RF=1:GOTO 159Ø
1720 POKE 198,Ø:RETURN
```

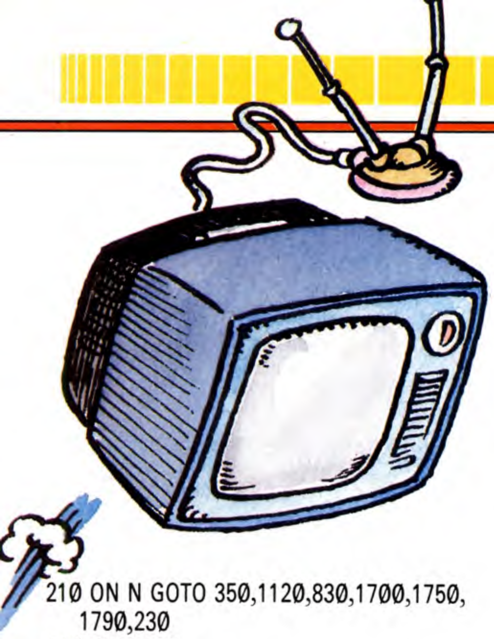
```
10 MODE4
20 PROCinitialise
25 REPEAT
30 PROCmenu
35 UNTIL N=55
40 MODE 6
50 END
60 DEF PROCinitialise
65 CR$=CHR$(13)+CHR$(1Ø)
70 DIM A$(2):A$(1)="Enter instruction: B-a
   blank or gap"+CR$+"W-a window or
   door or it will be a wall then enter the
   direction and distance in METRES; eg
   WU3R2 or D7L3 or R5."+CR$+"F to
   finish or C to clear the plan"
75 A$(2)="Enter the directions and distance
   in"+CR$+"CENTIMETRES : eg U5ØR3Ø;
```

```
   D6ØL2Ø or U7Ø.□□F to finish or C to
   clear the plan□□□□□□WARNING
   The first line is NOT drawn"
80 DIM h$(1Ø)
90 DIM take(2) ,put(3Ø,4) ,def(2Ø,9)
110 FOR P=1 TO 9 STEP 2
120 FOR T=1 TO 9
130 READ def(P,T),def(P+1,T)
140 NEXTT
150 NEXT P
160 DATA Ø.3,Ø.25,-Ø.6,Ø,Ø,-Ø.5,Ø.6,Ø,Ø,
   Ø.5,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
170 DATA Ø.255,Ø.255,-Ø.55,Ø,Ø,-Ø.55,
   Ø.55,Ø,Ø,Ø.55,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
180 DATA Ø.3,Ø.25,-Ø.6,Ø,Ø,-Ø.5,Ø.6,Ø,Ø,
   Ø.5,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
190 DATA Ø.5,Ø.25,-1,Ø,Ø,-Ø.5,1,Ø,Ø,Ø.5,
   Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
200 DATA Ø.25,Ø.255,-Ø.5,Ø,Ø,-Ø.55,Ø.5,
   Ø,Ø,Ø.55,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
210 FOR T=1 TO 1Ø
220 READ h$(T)
230 NEXTT
240 DATA cu,co,wa,si,fr,6,7,8,9,1Ø
250 i=Ø:j=Ø:flag=Ø:flag1=Ø:flag2=Ø:
   flag3=Ø:scale=158:G=Ø
260 ENDPROC
270 DEF PROCmenu
280 GCOLØ,1
290 VDU 28,Ø,31,39,26
300 VDU 24,Ø;199;1279;1Ø23;
310 CLS:PRINT"1. Plan room 2. Design layout"
320 PRINT"3. Design equipment 4. Save
   design"
330 PRINT"5. Load design 6.Print design"
340 PRINT"7. Exit program"
350 PRINT'"PRESS THE NUMBER OF YOUR
   CHOICE";
360 *FX15,Ø
370 N=GET
410 IF N<49 OR N>55 THEN 37Ø
420 IF N=49 PROCdrawroom
430 IF N=5Ø PROCselectequipment
```

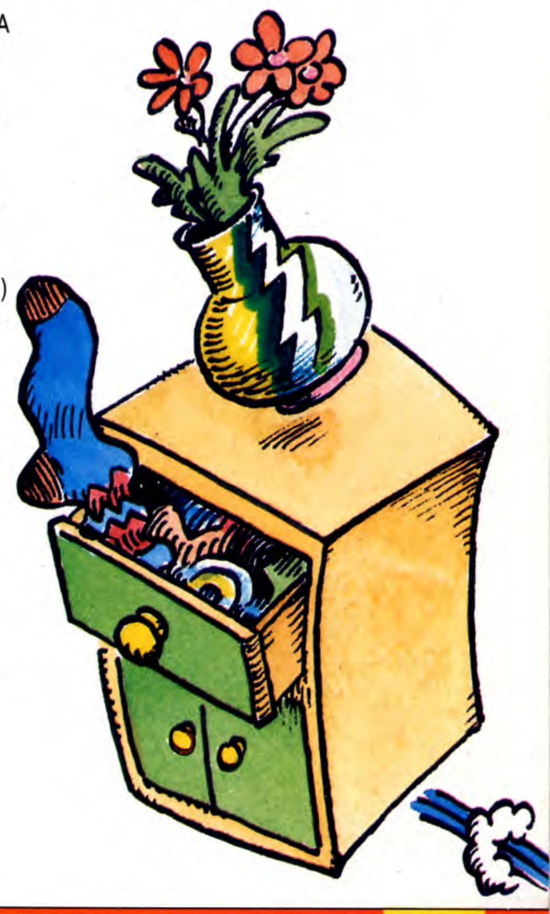


```
440 IF N=51 PROCownequipment
450 IF N=52 PROCsave
460 IF N=53 PROCload
470 IF N=54 PROCprint
490 ENDPROC
500 DEF PROCdrawroom
510 CLS:INPUT"Enter maximum length of room
   in METRES"length
520 VDU5:MOVE 44Ø,1Ø23:PRINT
   "LENGTH=";length;" METRES":VDU 4
530 scale=79Ø/length
540 PROClines(1)
550 ENDPROC
560 DEF PROClines(N)
570 LOCAL D$,D
580 IF N=1 THEN MOVE23Ø,2ØØ ELSE
   MOVE64Ø,61Ø
590 GCOLØ,1:PLOT65,Ø,Ø:CLS:flag=Ø
600 PRINTA$(N);
610 INPUTQ$
620 GCOLØ,1:T$=LEFT$(Q$,1):IF T$=
   "W" AND N=1 THEN flag=1:Q$=MID$
   (Q$,2)
630 IF N=1 AND T$="B" GCOLØ,Ø:Q$=
   MID$(Q$,2)
640 IF N=2 AND G=1 THEN GCOLØ,Ø
650 IF T$<>"C" THEN 69Ø
660 GCOLØ,Ø:MOVEØ,2ØØ:MOVE128Ø,2ØØ:
   PLOT85,Ø,98Ø:PLOT85,128Ø,98Ø:GCOLØ,1
670 IF N=2 THEN FOR T=1 TO 9:def(Z,T)
   =Ø:def(Z+1,T)=Ø:NEXT:G=1
680 GOTO 58Ø
690 IF T$="F" THEN CLS:ENDPROC
700 newx=Ø:newy=Ø:P=1
710 IF INSTR("UDLR",LEFT$(Q$,1))=Ø OR
   VAL (MID$(Q$,2))=Ø THEN PRINT
   "Please re-enter":GOTO 6ØØ
720 FOR I=1 TO 2
730 D$=MID$(Q$,1,1):D=VAL(MID$
   (Q$,2))
740 P=P+1+LENSTR$D
750 IF D$="U" newy=D
760 IF D$="D" newy=-D
770 IF D$="L" newx=-D
780 IF D$="R" newx=D
790 Q$=MID$(Q$,P):IF INSTR("UDLR",
   LEFT$(Q$,1))=Ø OR VAL
   (MID$(Q$,2))=Ø THEN I=2
800 NEXT
810 IF N=1 THEN 85Ø
820 IF G>9 PRINT'"Sorry only a maximum of
   8 sides."'"Press any key and then C if you
   want to start again":A$=GET$:GOTO 59Ø
830 PLOT1,nscale*newx,nscale*newy:def
   (Z,G)=newx/1ØØ:def(Z+1,G)=
   newy/1ØØ:G=G+1
840 GOTO 59Ø
850 IF flag PLOT 17,scale*newx,scale*newy□
   ELSE PLOT1,scale*newx,scale*newy
860 GOTO 59Ø
960 DEF PROCselectequipment
970 CLS
980 phi=Ø:c=scale:s=Ø
990 VDU 24,Ø;Ø; 1279; 1Ø23;
1000 GCOLØ,1
1010 VDU5:FOR I=1 TO 1Ø
1020 MOVE I*11Ø,1ØØ:IF h$(I)="" THEN
   PRINT;I:ELSE PRINTh$(I)
1030 NEXT:VDU 4
1080 GCOL3,1
1090 take(1)=32:take(2)=1ØØ
1100 PROCpoint(take(1),take(2))
1110 REPEAT
1120 *FX15,1
1130 PROCpoint(take(1),take(2))
1140 IF INKEY(-26)take(1)=take(1)-8
1150 IF INKEY(-122)take(1)=take(1)+8
1160 IF INKEY(-58)take(2)=take(2)+8
1170 IF INKEY(-42)take(2)=take(2)-8
1180 IF INKEY(-82)PROCselect
1190 IF INKEY(-56) AND i<>Ø
   PROCputdown
1200 IF INKEY(-66)PROCadjustment
1210 PROCpoint(take(1),take(2))
1220 UNTIL INKEY(-17)
1260 ENDPROC
1270 DEF PROCpoint(a,b)
1280 GCOL3,1
1290 VDU 5:MOVEa-32,b:IF i=Ø THEN VDU
   42 ELSE PRINTh$(i)
1300 VDU 4:ENDPROC
```

```
10 PCLEAR8:CLEAR 2ØØØ
20 DEF FNA(XM)=1.9*SC*XM
30 DIM O$(9),S(1Ø)
40 O$(Ø)="DR1ØØ;DU5Ø;DL1ØØ;DD5Ø;BR8;
   BU8;DR4Ø;DU34;DL4Ø;DD34;"
50 O$(1)="DR5Ø;DU6Ø;DL5Ø;DD6Ø;BR1Ø;
   BU1Ø;DR1Ø;DU1Ø;DL1Ø;DD1Ø;BR2Ø;
   DR1Ø;DU1Ø;DL1Ø;DD1Ø;BU2Ø;DU1Ø;
   DR1Ø;DD1Ø;DL1Ø;BL1Ø;DL1Ø;DU1Ø;
   DR1Ø;DD1Ø;BL2Ø;BU15;DR5Ø;"
60 O$(2)="DR1ØØ;DU6Ø;DL1ØØ;DD6Ø;"
70 O$(3)="DR3Ø;DU3Ø;DL3Ø;DD3Ø;DU2Ø;
   DR3Ø;"
80 O$(4)="DR6Ø;DU6Ø;DL6Ø;DD6Ø;"
90 CLS
100 PRINT@96,TAB(6)"1: PLAN ROOM"
110 PRINT TAB(6)"2: DESIGN LAYOUT"
120 PRINT TAB(6)"3: DESIGN EQUIPMENT"
130 PRINT TAB(6)"4: SAVE DESIGN"
140 PRINT TAB(6)"5: LOAD DESIGN"
150 PRINT TAB(6)"6: PRINT DESIGN"
160 PRINT TAB(6)"7: EXIT PROGRAM"
170 PRINT@422,"ENTER YOUR CHOICE";:
   INPUT N
180 IF N<1 OR N>7 THEN 9Ø
190 IF N=2 AND F1=Ø THEN CLS:
   PRINT"YOU MUST SELECT OPTION 1
   FIRST":SOUND 1,2Ø:GOTO 9Ø
200 IF N=1 THEN F1=1
```



```
210 ON N GOTO 350,1120,830,1700,1750,
    1790,230
220 GOTO 90
230 CLS:PCLS:END
240 RF=Ø:COLORØ
250 LINE(200,Ø)-(255,191),PSET,B
260 DRAW "BM206,10;S4;AØ;NR4D6R4BR2U
    6D3R4D3U6BR2D6R4U6NL4BR2D6R4U6N
    L4BR2NR4D3R4D3NL4BR2NR4U3NR2U3R
    4BR4BD2DBD2D"
270 IF RF=1 THEN RETURN
280 DRAW "BM203,30;D6R2NU3R2U6BR6R4
    D6L4U6BR10D6R4U3L4R3U3L3BR10NR4D
    3R4D3NL4BR2U6R4D3L4BR6U3NR4D6R4"
290 DRAW "BM218,43;D6U3NR2U3R4BR6NR
    4D6R4"
300 RETURN
310 RF=1:COLORØ,1:LINE(201,1)-
    (254,190),PRESET,BF:GOSUB 250
320 DRAW "BM208,30;ND6R4D3L4BR10BU3
    D6R3E1U4H1L3BR10;ND6R4D3L4R1F3BR
    6U6R4L4D3R2"
330 RETURN
340 DRAW "BM213,70;D4F1R2E1U4H1L2G1B
    U1BR8BD3R3BR5U3R4D3NL4D3BR8UBU2
    U2R2U2L4D2":RETURN
350 CLS
360 PRINT "MAX LENGTH OF ROOM
    (METRES)"
370 INPUT LE
380 IF LE>100 OR LE<3 THEN 350
390 SC=100/LE
400 PMODE 4,1:COLORØ,1:PCLS:SCREEN1,Ø
410 XM=Ø:YM=LE
420 GOSUB 240
430 XX=FNA(XM):YY=FNA(YM):IF PPOINT
    (XX,YY)=1 THEN PSET(XX,YY,Ø) ELSE
    PSET(XX,YY,1)
440 I$=INKEY$:IF I$="" THEN 430
450 OX=XM:OY=YM
460 IF I$="□" THEN COLOR Ø:GOTO 530
470 IF I$="B" THEN COLOR 1:GOTO 530
480 IF I$="C" THEN 400
490 IF I$="F" THEN FORK=1TO4:
    PCOPYK□TOK+4:NEXT: GOTO 90
500 IF I$="W" THEN 710
510 IF I$="O" THEN 800
520 GOTO 430
530 CLS
540 FOR A=1 TO 2
550 PRINT "DIRECTION";A;
    "□U/D/L/R□";:INPUTD$(A)
560 IF D$(A)="" THEN 600
570 PRINT "DISTANCE";A;:INPUT D(A)
580 IF INSTR(1,"UDLR",D$(A))=Ø THEN550
590 NEXT
600 SCREEN1,Ø:FOR A=1 TO 2
610 IF D$(A)="" THEN 670
620 IF D$(A)="L" THEN XM=XM-D(A)
630 IF D$(A)="R" THEN XM=XM+D(A)
640 IF D$(A)="U" THEN YM=YM-D(A)
650 IF D$(A)="D" THEN YM=YM+D(A)
660 NEXTA
670 IF XM<Ø OR XM>LE OR YM<Ø OR
    YM>LE THEN SOUND 1,2:XM=OX:
    YM=OY:GOTO 430
680 X1=FNA(OX):Y1=FNA(OY):X2=FNA
    (XM):Y2=FNA(YM)
690 LINE(X1,Y1)-(X2,Y2),PSET
700 GOTO 430
710 CLS:INPUT "DIRECTION U/D/L/R ";D$
720 INPUT "DISTANCE";D
730 X1=FNA(OX):Y1=FNA(OY)
740 POKE 178,2
750 IF D$="L" THEN XM=XM-D:X2=
    FNA(XM):LINE(X1,Y1)-(X2,Y1+3),PSET,
    BF
760 IF D$="R" THEN XM=XM+D:X2=FNA
    (XM):LINE(X1,Y1)-(X2,Y1+3),PSET, BF
770 IF D$="U" THEN YM=YM-D:Y2=
    FNA(YM):LINE(X1,Y1)-(X1+3,Y2),PSET,
    BF
780 IF D$="D" THEN YM=YM+D:Y2=
    FNA(YM):LINE(X1,Y1)-(X1+3,Y2),PSET,
    BF
790 SCREEN1,Ø:GOTO 430
800 CLS:INPUT "DIRECTION U/D/L/R "; D$(1)
810 INPUT "DISTANCE";D(1)
820 D$(2)="":COLOR1:GOTO 600
830 CLS
840 INPUT "ENTER NUMBER OF ITEM YOU
    WISH TODEFINE (Ø-9)";N
850 IF N<Ø OR N>9 THEN 830
860 O$(N)=""
870 PRINT"USE <SPACE> FOR
    LINE":PRINT"USE 'B' FOR BLANK
    LINE":FORD=1 TO 1000:NEXT
880 PMODE4,1:COLORØ,1:PCLS:SCREEN1,Ø
890 X=75:Y=145
900 DRAW "BM75,150;R100NG3NH3;BM70,
    145;U100NF3NG3"
910 IF PPOINT(X,Y)=Ø THEN PSET(X,Y,1)
    ELSEPSET(X,Y,Ø)
920 I$=INKEY$:IF I$="" THEN 910
930 OX=X: OY=Y
940 IF I$="C" THEN 830
950 IF I$="F" THEN 90
960 IF I$="□" THEN COLOR Ø
970 IF I$<>"□" THEN COLOR 1
980 IF$="□" THEN O$(N)=O$(N)+"D"
    ELSE O$(N)=O$(N)+"B"
990 CLS:INPUT "DIRECTION U/D/L/R ";D$
1000 IF INSTR(1,"UDLR",D$)=Ø THEN 990
1010 INPUT "DISTANCE (CMS)";D
1020 IF D<=Ø OR D>200 THEN 1010
1030 SCREEN1,Ø
1040 IF D$="L" THEN X=X-D/2
1050 IF D$="R" THEN X=X+D/2
1060 IF D$="U" THEN Y=Y-D/2
1070 IF D$="D" THEN Y=Y+D/2
1080 IF X<75 OR X>175 OR Y<45 OR
    Y>145 THEN SOUND 1,3:X=OX:Y=OY:
    GOTO 910
1090 LINE(OX,OY)-(X,Y),PSET
1100 O$(N)=O$(N)+D$+STR$(D)+";"
1110 GOTO 910
1120 FOR K=1 TO 4: PCOPYK+4TOK:
    NEXT:CLS
1130 PMODE4,1:SCREEN1,Ø
1140 GOSUB 310
1150 X=128:Y=96:RT=Ø
1160 IF X<3 THEN X=3
1170 IF Y<3 THEN Y=3
1180 GET(X-3,Y-3)-(X+3,Y+3),S,G
1190 DRAW "CØ;BM"+STR$(X)+",
    "+STR$(Y)+"NU3ND3NL3NR3"
1200 I$=INKEY$:IF I$="" THEN 1200
```



CLIFFHANGER: ROCKY 1

More and more troubles are being heaped on Willie. In the first part of this two-part article boulders are being prepared to roll down the slope at him

As if Willie's problems were not enough—what with the tide coming in, the goats eating his picnic, cliff-climbing to do and potholes and snakes to contend with—you are now going to start rolling rocks at him.

The process of putting a boulder at the top of the slope, moving it down the slope, animating it so that it looks like it is rolling, checking to see that it remains on the slope, blanking it out once it has moved, checking to see whether it has hit Willie, and losing it altogether once it has hit the sea is a complex one. So it is going to be handled in two parts.

The first part of the boulder-moving routine given here deals with moving the boulder down the slope. Unfortunately, it will not work without the other part of the routine, which deals with hitting Willie and starting the boulder off at the top of the slope. This will be given in the second part of this article. So, for now, key in and assemble this routine, but do not run it. Without part two it will crash.

```
     org 58993
bar  ld a,(57344)
     cp Ø
     jr z,blm
     cp 3
     jr z,blm
     ret
blm  ld a,(57358)
     cp 1
     jr z,bma
     ld hl,(57356)
     ld bc,5712Ø
     ld a,42
     call 58217
     inc hl
     ld a,45
     ld bc,15616
     call 58217
     ld hl,(57356)
     ld de,48Ø
     sbc hl,de
     jr z,bri
     ld hl,(57356)
     ld de,2256Ø
     add hl,de
     ld a,(hl)
     cp 15
     jr z,bri
     cp 45
     jr nz,bok
     ld hl,(57356)
     ld bc,15616
     ld a,45
     call 58217
     ld de,32
     add hl,de
     ld (57356),hl
     ld bc,5712Ø
     ld a,42
     call 58217
bok  ld hl,(57356)
     dec hl
     ld (57356),hl
     ld a,1
     ld (57358),a
     ret
     org 59137
bri  *
     org 59Ø97
bma  *
```

The first thing the rock-rolling routine has to do is to check whether the game is at a level where rock-rolling is needed. If you remember, Willie has to dodge flying boulders in level one and level four of Cliffhanger. The variable that corresponds to the game level is stored in memory location 57,344, Ø for level one, 3 for level four.

So the contents of memory location 57,344 are loaded into the accumulator and compared first with Ø, then with 3. If either of these two values are found the **jr z,blm** instruction jumps the process on into the main boulder-moving routine. But if neither are found, the processor reaches the **ret** and simply returns to the main routine.

IN THE MODE?

To animate the boulder there is data for two different boulder pictures stored in memory. When these are printed up on the screen alternately it gives the impression that the boulder is rolling.

To know which boulder picture to print on the screen the processor must know which picture was printed up last. The variable in



- CHECKING THE LEVEL
- TWO-FRAME ANIMATION
- CHECKING FOR LAND UNDERNEATH
- DROPPING DOWN THE SLOPE

57,358 tells it. Its contents are either Ø or 1. So the contents are loaded into the accumulator and compared to 1. If they are 1, the processor jumps to the **bma** routine which is given in the second part of this article. But if the contents of 57,358 are Ø, it continues with the routine given here.

You will notice that during the course of the two routines the variable in 57,358 is flipped back—if it was 1 it is flipped to Ø, if it was Ø it is flipped to 1—so next time the processor performs the boulder-moving routine it takes the other branch here and prints up the other boulder picture.



Naturally the contents of 57,358 are set to an initial value by the initialization routine which sets the game up.

ROCK ON

The position of the boulder is stored in memory location 57,356. So this is loaded up into HL. BC is loaded with 57,12Ø which is the start of the data for the boulder in its first position. And 42—which corresponds to red on cyan—is loaded into A.

The **print** routine, which is at 58,2Ø8, is then called. This—as always—prints up the data pointed to by the contents of BC in the colour specified by the contents of A in the position given by the contents of HL.

HL is then incremented to give the former position of the boulder. (For the moment you are only dealing with a rock rolling along a flat part of the slope. You'll see how the one-space drop is done to simulate it rolling over one of the diagonal parts of the slope subsequently.)

A is then loaded with 45—this corresponds to the colour cyan on cyan. And BC is loaded with 15,616. This location is in ROM and is the data for an empty space. Then **print** is called again. This blanks out the last rock so that you do not get a whole trail of them printed, nose to tail, down the screen.

ROCK OFF

Next you must check to see whether the rock has reached the extreme left-hand side of the screen. The far left of the roll is screen position 4ØØ. So the HL register is reloaded with the boulder's screen position from 57,356. It was incremented, remember.

DE is then loaded with 48Ø and the contents of the two registers are subtracted.

If the result is zero—that is the screen position of the boulder is 48Ø, or the final position of its roll—the instruction **jr z,bri** sends the processor to the **bri** routine. This prints over the boulder with a blank again and sets it back at the top of the slope again.

This routine is given in part two of this article and, unfortunately, this part will not work without it.

IN THE WATER?

But as the tide comes up the rock hits the water before it reaches the left-hand edge of the screen, you need to check for a splash. If it does hit the water, the boulder must then be switched off again. You wouldn't want it rolling along underwater. That wouldn't look at all realistic.

The simplest way to check whether the rock has hit the water is to check the attribute of the screen position directly underneath it. if its white on blue, the colour of the sea, then the boulder must be painted out. So HL is loaded with the boulder position from 57,356 yet again. And DE is loaded with 22,58Ø. There are, in fact, only 22,528 memory locations separating corresponding screen locations in the display file and the attribute file. But an extra 32 is added here so that the address of the attribute of the screen location one line down—or 32 character squares along—the screen is located.

The contents of HL and DE are then added. The HL and DE register pairs are always used for two-byte additions and subtractions and the result is always left in HL. So the contents of the memory location pointed to by HL—which is now the appropriate place in the attribute file—are loaded into the accumulator by the indirect instruction **ld a,(hl)**.

They are then compared with 15, which is white on blue—in other words, the sea colour. And if the sea is below the rock, the instruction **jr z,bri** sends the processor off to the routine in part two which paints out the boulder and starts it off from the top of the slope again.

THAT SINKING FEELING

Now that you have got the attribute of the character square below the boulder in memory it would be a shame to waste it. So the contents of the accumulator are compared to 45.

This, you remember, corresponds to cyan on cyan, the sky colour. Here you are checking to see if there is sky underneath the boulder.

If there is not, and the boulder is still

firmly on the ground, the **jr nz,bok** jumps the processor onto the **bok** which leaves the boulder where it is and simply tidies up the variables before returning. But if there is sky underneath the boulder, you had better do something about it and the processor continues then with the next part of the routine here.

If you remember, so far the boulder has only been moved to the left. Its vertical position has not been altered to take into account the fact that the slope falls away. So if there is sky beneath the boulder, that means that boulder has run over the edge of an inclined section by a character square. Thus, it needs to be blanked out and printed one character square lower. In machine code, all this is going to happen so fast that your eye—or the TV screen for that matter—will have no time to react. So you will not see the boulder





in its intermediate position flying gracefully through the air.

Once again the HL register is loaded with the current screen position of the boulder from 57,356. BC is loaded with the ROM data for a space from 15,616 as before. And A is loaded with 45—cyan on cyan. The **print** routine is called to blank out the boulder as before.

DE is then loaded with 32 which is added to HL. This moves the pointer in HL one character square down the screen. And the boulder position is updated by copying the pointer in HL back into memory location 57,356. It is important to remember that the **ld** instruction only copies the contents of one memory location or register into another memory location or register. The contents of its place of departure remain unchanged. So the new screen position is still in HL when BC is loaded with the start address of the first boulder picture data. A is loaded with 42—red on cyan—and the **print** routine is called. Then a red rock is printed in the new screen position, one character square down the screen from where the boulder appeared last time.

ROLLING STONES

Whether the rock has had to be reprinted one character square down the screen or not, the variables have to be reset to make the rock appear to roll down the screen.

So yet again HL is loaded with the current screen position in 57,356, though it might have been updated since the last time it was loaded up if the rock has rolled down an incline. The contents of HL are then decremented and the result is loaded back into 57,356. Next time the rock-rolling routine is called, this pointer will point one character square to the left.

When the processor decided to come down this branch of the routine and print up the first boulder picture, it was because the contents of the boulder mode location—57,358—was Ø. Next time you want it to print the other boulder picture. So 1 is loaded into the accumulator and then loaded back into 57,358.

Before you get into moving the boulder proper, this little routine takes a look at the character immediately under the boulder. It does this to check whether the slope is still



there—if it is not, the boulder will have to be moved down until it is on the slope—or it has reached the sea.

```
ORG  20736
LDA  $C008
CLC
ROR  A
STA  $0352
LDA  $C009
CLC
ROR  A
STA  $0353
JSR  $5000
LDA  ($FB),Y
RTS
```

The double density coordinates that you worked out before are stored in memory locations $C008 and $C009. The double density X coordinate in $C008 is loaded into the accumulator.

To convert this back into a regular coordinate it has to be divided by two. This is done simply by rotating the contents of the accumulator one place to the left. Note that the carry flag is cleared first. A rotate moves whatever is in the carry flag into the empty bit of the register. So if, for any reason, the carry flag was set you'd get a spurious result here. Clearing the carry flag here precludes this. The result is stored in $0352 which is the memory location used to pass the X coordinate into the multiplication routine in part nine of Cliffhanger (see page 1146).

The next four instructions take the double density Y coordinates through the same process. Again the carry flag is cleared before the rotation—it would have been set by the previous rotation if a bit had been shifted out of the least significant bit—and the least significant bit is ignored.

Note that the double density Y coordinate in $C009 is already pointing to the character square below the boulder, so it does not have to be adjusted now.

The multiplication routine at $5000 is then called. This, as you have seen, converts X and Y coordinates into a screen position which is stored in memory locations $FB and $FC.

The character displayed on the appropriate screen location is then loaded into the accumulator by LDA ($FB),Y. The Y is set to zero by the multiplication routine, but indexed indirect addressing is the only form of indirect addressing available. And this instruction takes as its base address the contents of memory locations $FB and $FC.

The processor then returns to the main boulder-moving routine, where it was called, with the character in the space below the boulder in the accumulator.

The first part of the boulder-moving routine given here contains a routine that looks at specific areas of the screen to check for collisions and one that prints the boulder up on the screen. Remember to set the computer up as normal before you key it in.

```
30 FORPASS=
 0TO3STEP3
40 P%=&1DBD
50 [OPTPASS
60 .Look
70 PHA
80 LDA#17
90 JSR&FFEE
100 PLA
110 CLC
120 ADC#128
130 JSR&FFEE
140 STX&70
150 STY&71
160 LDA#31
170 JSR&FFEE
180 TXA
190 LSRA
200 JSR&FFEE
210 LDA#64
220 SEC
230 SBC&71
240 LSRA
250 JSR&FFEE
260 LDA#&87
270 JSR&FFF4
280 TXA
290 AND#31
300 LDX&70
310 LDY&71
320 RTS
330 .PtBarrel
340 LDX&78
350 LDY&79
360 JSR&1964
370 LDA#5
380 JSR&FFEE
390 LDA#18
400 JSR&FFEE
410 LDA#3
420 JSR&FFEE
430 LDA#3
440 JSR&FFEE
450 LDX#242
```

```
460 LDA&75          520 JSR&FFEE
470 AND#1           530 LDA#4
480 BEQLb1          540 JSR&FFEE
490 LDX#243         550 RTS
500 .Lb1            560 ]
510 TXA             570 NEXT
```

LOOK AND LEARN

When you look at a character on the screen, its background colour must be the same as the one the machine is currently using—otherwise your computer would not be able to recognizé what the bit pattern is. It would not know which is foreground and which is background.

So when you enter this routine you have to specify the background colour of the character square you're looking at. This has to be put in A, and the X and Y coordinates of the screen position you want to look at in the X and Y registers.

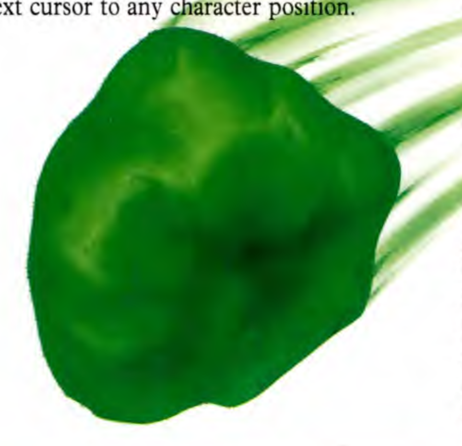
The background colour in A is pushed onto the stack immediately the processor enters the routine. The accumulator has to be used for something else for a moment. It has to be loaded with 17 which is then output to the screen routine at FFEE. This gives a VDU 17 which changes the background colour.

The colour specified is then pulled back off the stack and 128 is added—it's the background colour you're changing, remember. The result of the addition is in A so it can be output directly by jumping to FFEE.

POSITIONING

The X and Y coordinates of the position you want to tab to are stored temporarily in zero page memory locations &7Ø and &71. They can then be manipulated easily without their values being lost. And that is important as you are going to have to work out the machine's own coordinates in this particular mode from the coordinates the program has been using.

A is loaded with 31 and output to the routine at FFEE. This allows you to move the text cursor to any character position.



The contents of the X register are then transferred into A and logically shifted to the right. This divides them by two, giving the correct coordinate for this mode.

To work out the appropriate Y coordinate the number in &71 has to be subtracted from 64 and the result divided by two, again by a logical shift right. This is output by the instruction in Line 25Ø.

READING THE SCREEN

An operating system call is used to read the character on the screen. If you look in your manual you will see that OSBYTE &87 reads the character at the text cursor position.

The text cursor has already been positioned by the tab routine above. And to get OSBYTE &87 you load the accumulator with &87 and jump to the subroutine at FFF4.

This returns the value of the screen character in the X register, so it has to be transferred into A before it can be manipulated.

The characters you are looking for with this routine are UDGs. The way these have been encoded means that you are only interested in bits zero to four. So the character value is ANDed with 31.

When the processor comes out of the routine it carries the character value of the five least significant bits in A. And the values in X and Y are restricted by loading them up from &7Ø and &71.

GETTING BOULDER

The current coordinates of the boulder are stored in &78 and &79. These are loaded into the X and Y registers. The processor then jumps to the subroutine at &1964. This routine was given in an earlier part of Cliffhanger and moves the graphics cursor to the coordinates given by the X and Y registers.

A is then loaded with 5 and the routine at FFEE is jumped to again. This gives a VDU 5 which allows you to write text to the graphics cursor. One of the advantages of this is that it allows you to superimpose what you write on what is already on the screen.

A is then loaded with 18 and FFEE is called again. This gives a VDU 18 or GCOL. The next four instructions—in Lines 41Ø to 44Ø—output two 3s to FFEE. These are the GCOL's two parameters. So this instruction gives a GCOL 3,3, which sets the colour of the boulder. The first 3, you note, means that the logical colour in the second parameter is Exclusively ORed with what is on the screen already. This means that this routine can be used to rub out a boulder was well, if it is called again in the same place.

Then the UDG number of one of the two boulder characters—242—is loaded into X.

Memory location &75 carries the direction variables for the boulder. Bit one carries a flag which tells the processor which boulder picture to print on the screen. There are two of them which are printed alternately to give the impression that the boulder is rolling.

This flag is loaded up into the accumulator and ANDed with 1. This effectively flips the flag—if it was 1 it flips it to Ø and if it was Ø it flips it to one.

If the result of this operation was zero, the BEQ instruction branches the processor over the next generation. This leaves the UDG number in X as it was and goes ahead, ready to output it. If not, X is loaded with the UDG number of the other boulder picture—243—instead. The flipping of the flag each time this routine is executed ensures that the two boulders are printed up in rotation.

Whichever UDG number is in X, it is then transferred into A and output to the screen by jumping to the subroutine at FFEE. This actually prints the boulder on the screen!

A is then loaded with 4 and output by jumping to FFEE. This gives a VDU 4—which is the opposite of a VDU 5—this separates the text and graphics cursors again and returns them to their normal function. Then the processor exits the routine.

TESTING

This routine is called by the routine coming in part two of this article. So it doesn't do anything much on the screen. But it can be tested by keying in the following instructions:

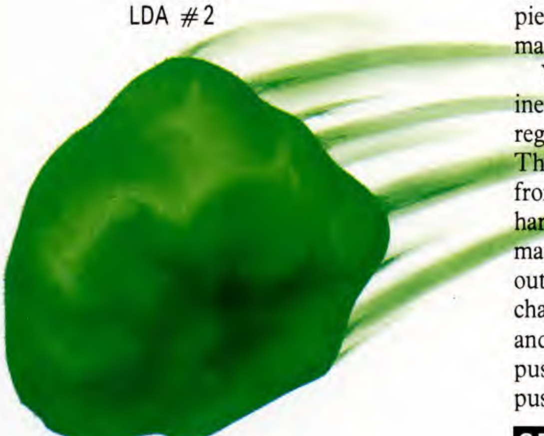
```
MODE 5: PRINT"A"
A%=Ø:X%=Ø:Y%=64
PRINT USR(&1DBD)AND&FF
```

This should print a 1 on the screen. And if you repeat the instructions with a B instead of an A, it should print a 2.

The first part of the boulder-moving routine given here deals with moving the boulder and checking for collisions. Unfortunately, it will not work without the other part of the routine, which deals with actually printing the boulder on the screen. This will be given in the second part of this article. So, for now, key in and assemble this routine, but do not run it. Without part two it will crash.

```
        ORG 19781
BAR     LDA 18238
        BEQ BLM
        CMPA #3
        BEQ BLM
        RTS
```

```
BLM     LDX 18253
        LDU #1536
        PSHS X
        JSR CHARPR
        PULS X
        LEAX -1,X
        CMPX #5344
        BEQ BRI
        STX 18253
        LDA ,X
        CMPA #$AA
        BEQ BRI
        CMPA #$55
        BEQ BNH
        CMPA #$5Ø
        BEQ BNH
        LDA #2
```



```
        STA 18252
BNH     LEAX 289,X
        LDA ,X
        CMPA #$AA
        BEQ BRI
        CMPA #$55
        BNE BOK
        LEAX -33,X
        STX 18253
CHARPR  EQU 194Ø2
BOK     EQU 19861
BRI     EQU 19894
```

The first thing the rock-rolling routine has to do is to check whether the game is at a level where rock-rolling is needed. If you remember, Willie has to dodge flying boulders in level one and level four of Cliffhanger. The variable that corresponds to the game level is stored in memory location 18,238. If Ø, you are on level one, if 3, you are on level four.

So the contents of memory location 18,238 are loaded into the accumulator. If they are Ø, the BEQ instruction will make the processor branch directly into the main rock-rolling routine. If not, they are compared with Ø. If that is found, again, the processor will branch. But if neither are found, the processor reaches the RTS and returns to the main routine.

BLANKING THE BOULDER

Memory location 18,253 is a variable which contains the current screen position of the boulder. That position is loaded into the X register. And U is loaded with the number 1536. U is the user stack pointer, so the area of memory from 1536 upwards becomes to all intents and purposes the user stack. Memory location 1536 is part of the screen memory—in fact, part of the sky. The processor then jumps to the CHARPR subroutine.

This subroutine prints a character on the screen, remember. It takes the data from the user stack and prints it up a byte at a time in the character square pointed to by X. So a piece of sky data is printed over the boulder, making it look like any other piece of sky.

You'll notice that before the CHARPR routine was jumped too, the contents of the X register were pushed onto the hardware stack. This does not mean that they were removed from X. They were simply copied onto the hardware stack and preserved there. CHARPR may interfere with the X register as it prints out the eight bytes of data which make up one character. So if you are calling a subroutine and need to preserve data in a register, always push it onto the stack. The rule is: if in doubt, push it. It saves a lot of crashes.

GETTING EDGY

To get the screen position back again, it simply has to be pulled back off the hardware stack. LEAX −1,X then decrements the contents of X which points it one screen position further left. This is then compared with 5344, which is the location of the edge of the screen where the boulder hits it when coming down.

If X is 5344 and the boulder has reached the edge of the screen, the BEQ BRI instruction branches the processor to the BRI routine which moves the boulder back up to the top of the slope.

But if X is not 5344 and the boulder has not reached the edge of the screen yet, the STX 18253 stores the next screen position in X—which is one character square to the left of the old one—back into the boulder position variable at 18253. This is the position the new boulder is going to be printed in.

WILLIE OR WON'T HE?

The next thing that is checked for is to see whether the boulder is about to hit Willie. So before the new boulder is printed up the contents of the screen position pointed to by the contents of the X register are loaded into the accumulator by the LDA ,X instruction.

This is then compared to \$55, the code for yellow, the sky colour, and \$5D, the snake's tongue. If the character square does contain \$55, or \$5D, sky, the processor skips the next two instructions. But if it doesn't, it must have hit Willie—he is the only thing that might stand in the way of a rolling boulder—the branch is not made. The accumulator is loaded with 2 and this is stored into memory location 18,253, the so-called die variable which is checked later to see whether Willie is dead.

A BIGGER SPLASH

The next thing that is checked for is to see if the rock has reached the surface of the water. If it has, you don't want to print a rock there, another one has to be started off from the top of the slope.

So X is incremented by 289, to move it onto the beginning of the next character square below—289 is $32 \times 8 + 32 + 1$. To count down the screen one character square you need to count along the screen memory eight lines of 32 bytes. The boulder actually floats one pixel line above the earth—and the sea—to give it clearer definition, so you need to count an extra 32 bytes along the screen memory. Then remember, you subtracted 1 to move the screen pointer one place to the left above, so that extra +1 compensates.

The contents of that character square are loaded up into the accumulator again by LDA ,X and compared to \$AA, which is the code for blue, the sea colour. If the sea is in that character square, BEQ BRI branches the processor off to the routine which starts the boulder back at the top of the slope again.

GROUNDED

While you are examining the contents of the character square beneath the boulder position, you might as well check to see whether they are sky or not. So they are compared to \$55, the sky colour again.

If no sky is found underneath the boulder position—in other words, it's earth—the BNE BOK instruction branches to the BOK routine which actually prints up the boulder.

But if there is sky under the boulder, it has to be dropped down one character square—as well as moving it one character square to the left. Remember now that the screen pointer is pointing to one line of pixels below the bottom of the last boulder position. So to move it back up one line of pixels you must subtract 32. And to move it one place to the left from there another 1 has to be subtracted. This is done by the instruction LEAX −33,X.

The processor then proceeds directly into the BOK routine to print the boulder in the appropriate place. This routine, along with the BRI routine, is given in the next part of Cliffhanger.

WARGAMING: OF MAPS AND MEN

Plan the map for Cavendish Field. Dot the battlefield with hills, forests and villages. And make the two warring factions draw up at their starting positions

In part one of this series of articles about writing computer wargames you set up the symbols for each of the military units needed for Cavendish Field. These symbols will be displayed (and moved) on a map, to show you the progress of the game, and allow you to plan your strategy.

PROGRAMMING THE MAP

Cavendish Field has an array to represent the map. The map will be displayed continuously in the game, taking the largest part of the screen display.

As the map is displayed all the time, strictly speaking, the array is unnecessary—there will always be an area of memory holding the screen which contains all the map's information. But it is worthwhile setting up the array, too, despite the memory sacrifice, because it can be read and written to with ease, rather than trying to POKE the information directly into areas of the screen memory.

The array has as many elements as there are screen positions within the map area. The Spectrum map is 30 by 16; the Commodore's map is 38 by 17; the Acorn's is 38 by 20; and the Dragon/Tandy's is 30 by 16.

The Acorn program runs in MODE 1, which alone consumes around 20K of memory. This leaves little memory left over for the program and the large map array—a normal array needs about five bytes per element, and an integer array needs about four bytes per element.

There simply isn't enough room to use an array of DIMensions 30 by 16, as dictated by the program design, so you need to use a new type of array—a byte array. This is simply a block of bytes determined by a DIMension statement.

The byte array used in Cavendish Field is a single-dimensional array and will need 760 elements, saving roughly 2K of memory over an integer array. Unfortunately, you'll need some more program lines to read from and write to a byte array.

THE TROOPS

To keep track of where the units are on the map—so that they can be moved, and to check if there are any obstacles in the way—you need another array, the troop array. But this array will be used for much more than just holding the position of the troops. In fact, it can hold every piece of information about them.

The troop array in Cavendish Field also holds information needed for combat, morale, movement and so on—exactly what is held in the troop array in any other game based on this structure will depend on the nature of the game you are writing, and each of these areas will be covered fully as you progress through the program.

SETTING UP THE ARRAYS

The following routine DIMensions the map and troop arrays:

```
350 DIM m(16,30)
355 DIM T(16,9)
```

```
350 DIM M(38,16)
355 DIM T(15,8)
```

```
25 DEF FNmread(x,y)=?(map%+(x*20)+y)
350 DIM map%□760
355 DIM T%(15,8)
430 DEF PROCmput(x,y,val)
440 ?(map%+x*20+y)=val
450 ENDPROC
```

```
350 DIM M(16,30)
355 DIM T(16,9)
```

All the programs simply DIMension the map array and the troop array. The map array is just the size of the screen display (except on the Acorns which work differently, as described above). The troop array is DIMensioned to hold 9 elements of information about each of 16 troop units. Where the numbers used are 15 and 8, the zero element of the array is being used, too.

In addition, the Acorn program DEFines a PROCedure to put values in the troop array, and a FuNction to read information from the troop array.

FILLING THE MAP

The next step is to determine the terrain, and the starting position of each unit and to display them on the screen. You could choose to set up a fixed map—if you wanted to try to duplicate a famous battle, you might decide to take this course. However, in most cases you will want to have a variety of maps open to you. The simplest way is to use your machine's random number generator.

Determining the terrain could involve making a simple random plot of a number of the terrain symbols—forest, hills and village. The problem with a simple random choice is that hills and woods are generally not dotted all over the place, but tend to concentrate in clumps. The program needs to take account of this.

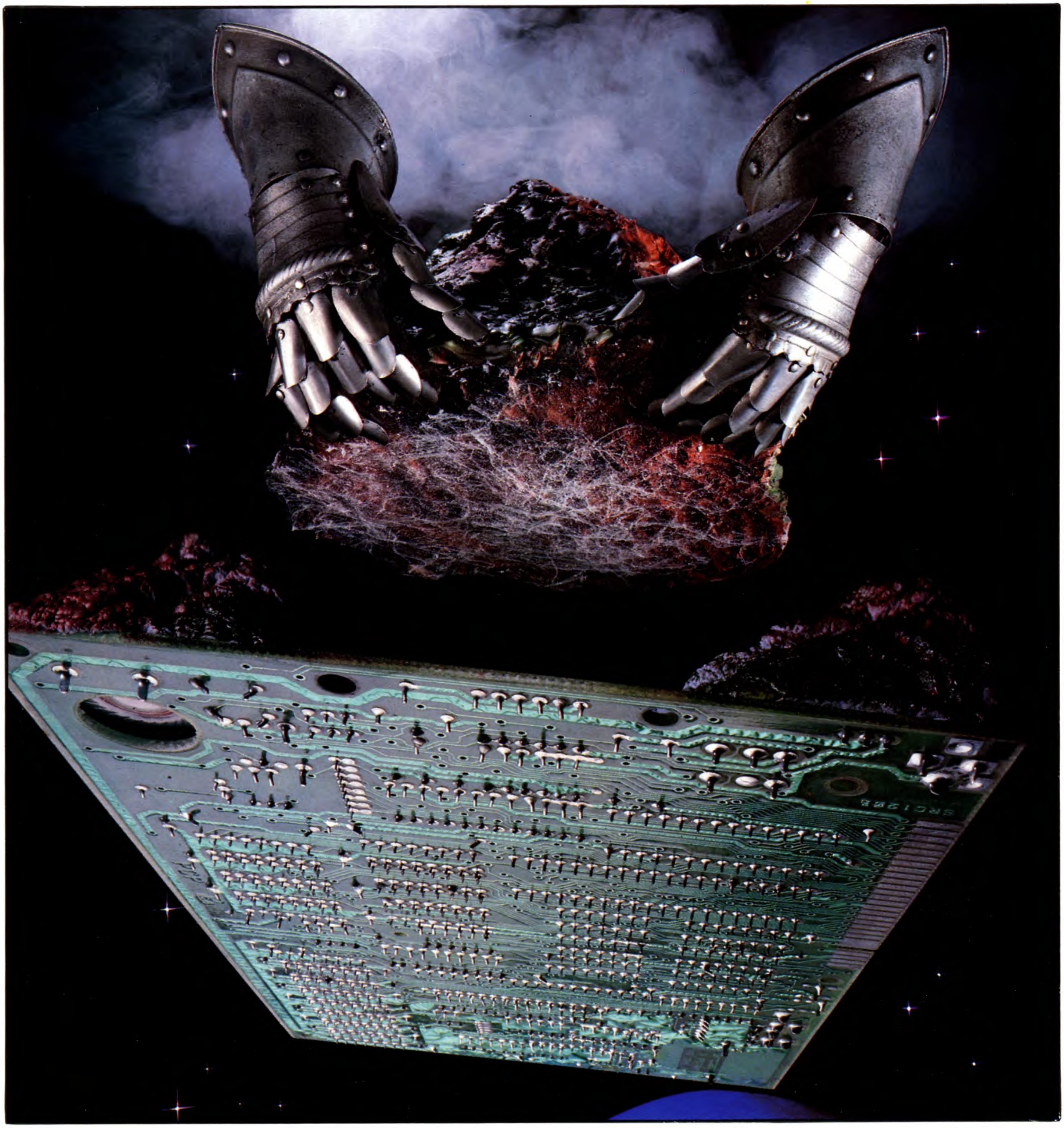
Another consideration when starting to set up a map for a wargame is the kind of terrain in which you would expect your kind of battle to take place. For example, as Cavendish Field is a medieval wargame you would expect it to take place in fairly open terrain. In this case, you wouldn't aim to place too many hills and forests on the map.

CHOOSING TERRAIN

The routine which chooses terrain is essentially random, but there is a degree of control over the selection. This ensures that the terrain is drawn realistically, with hills and woods clumped together.

```
20 DEF FN r(x)=INT (RND*x)+1
800 REM Choose Terrain
810 LET R=FN r(50)
820 IF R>5 THEN LET R=0
830 IF R>4 THEN LET R=3: RETURN
840 IF R>1 THEN LET R=2
850 RETURN
```

```
800 REM CHOOSE TERRAIN
810 R=FNR(50)
820 IF R>5 THEN R=0
830 IF R>4 THEN R=3:RETURN
840 IF R>1 THEN R=2
850 RETURN
```







```
800 DEF PROCchter
810 R=RND(5Ø)
82Ø IF R>5 THEN R=Ø
83Ø IF R>4 THEN R=3:ENDPROC
84Ø IF R>1 THEN R=2
85Ø ENDPROC
```

```
8ØØ REM CHOOSE TERRAIN
81Ø R=RND(5Ø)
82Ø IF R>5 THEN R=Ø
83Ø IF R>4 THEN R=3:RETURN
84Ø IF R>1 THEN R=2
85Ø RETURN
```

Integer random numbers are used throughout Cavendish Field. The Acorn, Dragon and Tandy machines can generate numbers in this form, but the Spectrum and Commodore cannot. Line 2Ø of the Spectrum program, and Line 195 of the Commodore program DEFines a FuNction to generate integer random numbers.

In each case, a random number, R, between 1 and 5Ø is generated. The Choose Terrain routine may change the value of R according to the value generated. If the generated value is greater than five, R is set to zero, the code for plains. If the generated value is five, R is set to three, the value representing hills; and if the value is two, three or four, R is set to two, the value representing forest. The remainder of cases are when R equals one—representing villages.

UNDER CONTROL

The Choose Terrain routine is called for each element along one dimension of the array, giving a random choice for the first terrain elements on the map. However, as was mentioned earlier, total randomness is not at all desirable. The next routine makes sure there is some pattern to the terrain upon which battle is to be fought.

```
37Ø LET i$="NWSE"
47Ø REM Create
48Ø FOR i=1 TO 16: GOSUB 8ØØ: LET
    m(i,1)=R: NEXT i
49Ø FOR i=1 TO 16
5ØØ FOR j=2 TO 3Ø
51Ø LET s=FN r(1Ø)
52Ø IF s<8 THEN GOSUB 8ØØ
525 IF s>=8 THEN LET R=m(i,j-1)
53Ø LET m(i,j)=R
54Ø IF R=3 AND j<3Ø THEN LET
    m(i,j+1)=4
55Ø IF m(i,j)<>Ø AND m(i,j)
    <>3 THEN PRINT AT i,j;CHR$
    (m(i,j)+143)
555 IF m(i,j)=3 AND j<>3Ø
    THEN PRINT AT i,j;CHR$ 146;AT i,j+
    1;CHR$ 147
56Ø NEXT j
57Ø NEXT i
58Ø GOSUB 72Ø
59Ø FOR i=1 TO 8
6ØØ FOR j=1 TO 2: LET T(i,j)=2: LET
    T(i+8,j)=2: NEXT j
61Ø FOR j=3 TO 4: READ T(i,j): LET
    T(i+8,j)=T(i,j): NEXT j
62Ø READ mr
63Ø FOR j=Ø TO 8 STEP 8
64Ø LET T(i+j,5)=mr+FN r(2)
65Ø LET T(i+j,6)=(FN r(1ØØ)*1Ø)+1Ø
66Ø LET T(i+j,7)=T(i+j,6)
67Ø NEXT j
68Ø LET T(i,8)=15
69Ø LET T(i+8,8)=1
7ØØ NEXT i
71Ø RETURN
```

```
37Ø I$="NWSE"
47Ø REM CREATE
48Ø FOR I=Ø TO 16:GOSUB8ØØ:M(Ø,I)=
    R:NEXT I
49Ø FOR I=Ø TO 16
5ØØ FOR J=1 TO 37
51Ø S=FNR(1Ø)
52Ø IFS<8 THEN GOSUB 8ØØ
525 IF S>=8 THEN R=M(J-1,I)
53Ø M(J,I)=R
54Ø IF R=3 AND J<37 THEN
    M(J+1,I)=4
55Ø IF M(J,I)<>Ø THENP=J:Q=I:
    GH=(M(J,I)+63):CL=Ø:
    GOSUB26ØØ
555 IF M(J,I)=3 THENP=J+1:Q=I:GH=
    (M(J,I)+64):CL=Ø:
    GOSUB26ØØ
56Ø NEXT J
57Ø NEXT I
58Ø GOSUB 72Ø
59Ø FOR I=Ø TO 7
6ØØ FOR J=Ø TO 1:T(I,J)=1:T(I+8,J)=
    1:NEXT J
61Ø FOR J=2 TO 3:READ T(I,J):T(I+8,J)=
    T(I,J):NEXTJ
62Ø READ MR
63Ø FOR J=Ø TO 8 STEP 8
64Ø T(I+J,4)=MR + FNR(2)-1
65Ø T(I+J,5)=(FNR(1ØØ)*1Ø)+1Ø
66Ø T(I+J,6)=T(I+J,5)
67Ø NEXT J
68Ø T(I,8)=16
69Ø T(I+8,8)=Ø
7ØØ NEXT I
71Ø RETURN
```

```
370 dir$="NWSE"
470 DEF PROCcr
480 FORi=ØTO19:PROCchter:
    PROCmput(Ø,i,R):NEXT
490 FORi=ØTO19
500 FORj=1TO37
510 s=RND(1Ø)
520 IF s<8 THEN PROCchter ELSE
    R=FNmread(j-1,i)
530 PROCmput(j,i,R)
540 IFR=3 AND j<37 THEN
    PROCmput(j+1,i,4)
550 IF FNmread(j,i)<>Ø AND FNmread
    (j,i)<>3 THEN PRINT TAB(j+1,i+1);
    CHR$(FNmread(j,i)+223) ELSE IF
    FNmread(j,i)<>Ø PRINT TAB(j+1,i+1);
    CHR$(227);CHR$(226)
560 NEXT:NEXT
580 PROCborder
590 FORi=ØTO7
600 FORj=ØTO1:T%(i,j)=1:T%(i+8,j)=1:
    NEXT
610 FORj=2TO3:READ T%(i,j):T%
    (i+8,j)=T%(i,j):NEXT
620 READ mo
630 FORj=ØTO8STEP8
640 T%(i+j,4)=mo+RND(2)-1:T%
    (i+j,5)=(RND(1ØØ)*1Ø)+1Ø:T%
    (i+j,6)=T%(i+j,5)
670 NEXT
680 T%(i,8)=2Ø:T%(i+8,8)=Ø
700 NEXT
710 ENDPROC
```

```
370 I$="NWSE"
470 REM CREATE
480 FOR I=1 TO 16:GOSUB 8ØØ:
    M(I,1)=R:NEXT I
490 FOR I=1 TO 16
500 FOR J=2 TO 3Ø
510 S=RND(1Ø)
520 IF S<8 THEN GOSUB 8ØØ
525 IF S>=8 THEN R=M(I,J-1)
530 M(I,J)=R
540 IF R=3 AND J<3Ø THEN M(I,J+1)=4
550 IF M(I,J)<>Ø AND M(I,J)<>3
    THEN LINE (J*8, I*8)- (J*8+7, I*8+7),
    PRESET, BF: DRAW "BM"+ STR$
    (J*8)+ ","+ STR$ (I*8)+ UC$ (M(I,J))
555 IF M(I,J)=3 AND J<>3Ø THEN
    DRAW "BM"+ STR$ (J*8)+ ","+
    STR$ (I*8)+ UC$(3)+ "BM"+ STR$
    ((J+1)*8)+ ","+ STR$ (I*8)+ UC$(4)
560 NEXT J,I
580 GOSUB 72Ø
590 FOR I=1 TO 8
600 FOR J=1 TO 2:T(I,J)=2:T(I+8,J)=2:
    NEXT J
610 FOR J=3 TO 4:READ T(I,J):T(I+8,J)=T
    (I,J):NEXT J
620 READ MR
630 FOR J=Ø TO 8 STEP 8
640 T(I+J,5)=MR+RND(2)
650 T(I+J,6)=(RND(1ØØ)*1Ø)+1Ø
660 T(I+J,7)=T(I+J,6)
670 NEXT J
680 T(I,8)=15
690 T(I+8,8)=1
700 NEXT I
710 RETURN
```

The routine generates a new random number, S, for each subsequent element of the array. The value of S may range from one to ten—chosen in Line 51Ø. Line 52Ø ensures that, for seven tenths of the time, the new element in the array will have randomly generated terrain—if S<8, then the program jumps to the Choose Terrain routine. For the other three tenths of the time, the element will have exactly the same terrain as that immediately to its left. This has the effect of creating blocks on the map. Lines 55Ø to 57Ø display the chosen terrain on the map.

DEPLOYING THE TROOPS

Troop positions are held in the troop array as a pair of coordinates—horizontal and vertical.

The starting positions of the opposing sides' units in Cavendish Field are at different ends of the map—the player's starting at the southern end (the very bottom of the screen display), and the computer's starting at the northern (the top of the display). The vertical coordinate, then, doesn't need to be chosen.

The horizontal coordinate, needs to be selected much like choosing the terrain. It needs to have an element of randomness, but some constraints have to be imposed. You must ensure that two or more units do not appear on the same square, for example.

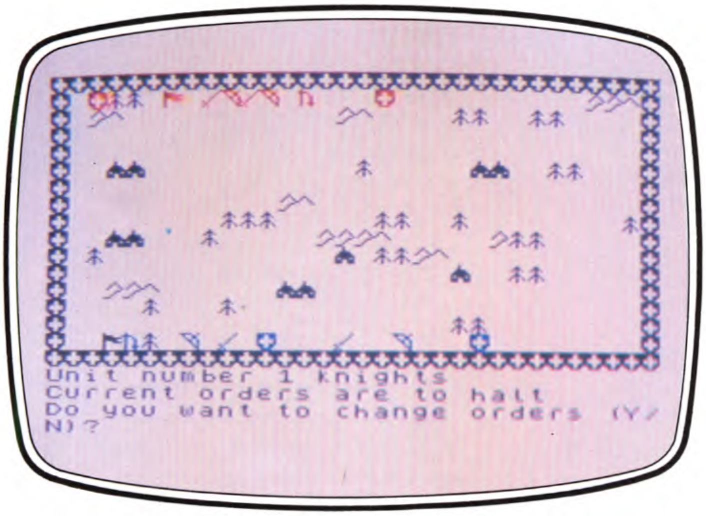
The following routine chooses the starting positions of each side. First it picks the troop units at random. Then it divides the map vertically into eight columns. Each of these represents the limits within which one of the units on each side will be placed. The actual print position is selected randomly within the limits of the column width.

```
860 REM Dispose Troops
870 INK 2
880 FOR m=1 TO 2
890 LET s=1: LET r=1
900 FOR k=1 TO 8
910 REM Dummy for Repeat loop
920 LET s=FN r(8*m)
930 IF T(s,9)<>Ø THEN GOTO 91Ø
940 LET r=FN r(4)+r
950 LET r=r-INT (r/3Ø)
960 LET T(s,9)=r
970 INK m
980 PRINT AT T(s,8),T(s,9);u$(s)
990 NEXT k
1000 NEXT m
1010 RETURN
```

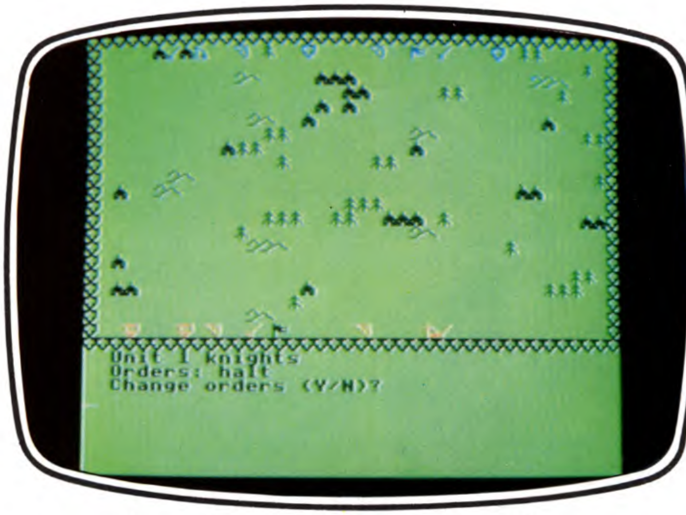
```
860 REM DISPOSE TROOPS
870 CL=1
880 FOR M=1 TO 2
890 S=Ø:R=Ø
900 FOR K=Ø TO 7
910 REM DUMMY FOR REPEAT LOOP
920 S=FNR(8*M)-1
930 IF T(S,7)<>Ø THEN 91Ø
940 R=FNR(6)+R
950 R=R-INT(R/37)
960 T(S,7)=R
970 IFM=2 THEN CL=9
980 P=T(S,7):Q=T(S,8):GH=VAL
    (MID$(U$,S+1,1))+67:GOSUB26ØØ
990 NEXT K
1000 NEXT M
1010 RETURN
```

```
860 DEF PROCds
870 COLOUR 2
880 FOR m=1TO2
890 s=Ø:r=Ø
900 FORk=ØTO7
910 REPEAT
920 s=RND(8*m)-1
930 UNTIL T%(s,7)=Ø
940 r=RND(6)+r
950 r=r □MOD 37
960 T%(s,7)=r
970 COLOUR m
980 PRINT TAB(T%(s,7)+1,T%(s,8)+1);
    MID$(unst$,(s□MOD 8)+1,1)
990 NEXT:NEXT
1010 ENDPROC
```

```
860 REM DISPOSE TROOPS
870 COLOR 2
880 FOR M=1 TO 2
890 S=1:R=1
900 FOR K=1 TO 8
910 REM
920 S=RND(8*M)
930 IF T(S,9)<>Ø THEN 91Ø
940 R=RND(4)+R
950 R=R-INT(R/3Ø)
960 T(S,9)=R
970 COLOR M:IF M=1 THEN COLOR 3
980 DRAW"BM"+STR$(T(S,9)*8)+","+
    STR$(T(S,8)*8):UU=VAL(MID$(U$,S,1)):
```



Cavendish Field shown on the Spectrum



The battle zone on the Acorn machines

```
A$=UC$(UU):GOSUB 3000
990 NEXT K
1000 NEXT M
1010 RETURN
```

Because each of the eight units is chosen at random, this helps each game to appear different, as a given unit doesn't always start in the same eighth of the map. As both armies are stored in the same array, the same routine can be used to choose positions for both armies—all you need is the FOR ... NEXT loop between Lines 880 and 1000. The loop also ensures that the two armies appear in different colour.

ON TO THE BATTLEFIELD

Once the starting positions of the two armies have been determined, the units can be made ready for display on screen.

```
410 LET U$=CHR$148+CHR$149+CHR$
   150+CHR$150+CHR$151+CHR$151
   +CHR$152+CHR$152:LETU$=U$+U$
```

```
410 U$="12334455":U$=U$+U$
2600 REM DRAW
2610 POKE (1064+P+(Q*40)),GH
2620 POKE (55336+P+(Q*40)),CL
2630 RETURN
```

```
410 unst$=CHR$(228)+CHR$(230)+
   CHR$(231)+CHR$(231)+CHR$(232)+
   CHR$(232)+CHR$(229)+CHR$(229)
```

```
410 U$="65778899":U$=U$+U$
```

In Line 410 a string, U$ (or unst$, in the case of the Acorn) is set up. This holds the code for the symbol representing each unit.

The Commodore needs to POKE directly on to the screen, rather than use the PRINT AT or PRINT TAB that are available on the Spectrum or Acorn machines. You need two POKEs, in fact, one to the screen position and one to colour memory. Both POKEs require the same parameters, but of course, have different addresses.

Lines 2600 to 2630 of the Commodore program POKE the information to the correct addresses.

BORDERING ON HOSTILITY

The screen display can be made more attractive, and less misleading if a border is drawn round the map:

```
720 REM Decorative Border
730 FOR i=0 TO 16
740 PRINT AT i,0;CHR$ 150;AT i,31;CHR$
   150
750 NEXT i
760 FOR i=0 TO 31
770 PRINT AT 0,i;CHR$ 150;AT 16,i;CHR$
   150
780 NEXT i
790 RETURN
```

```
720 REM DECORATIVE BORDER
730 FOR I=0 TO 39
740 POKE (1024+I),70:POKE
   (1024+I+(40*18)),70
750 NEXT I
790 RETURN
```

```
720 DEF PROCborder
730 FORi=0TO39
740 PRINT TAB(i,0);CHR$(231);TAB(i,22);
   CHR$(231)
750 NEXT
760 FORi=0TO22:PRINT TAB(0,i);CHR$
   (231);TAB(39,i);CHR$(231):NEXT
790 ENDPROC
```

```
720 REM BORDER
730 LINE(0,128)-(255,135),PRESET,BF
740 LINE(248,0)-(255,191),PRESET,BF:LINE
   (4,4)-(252,132),PSET,B
790 RETURN
```

The routines simply draw a series of symbols (the same as used for one of the troop units) round the map.

MOBILIZING YOUR FORCES

Now that the map is set up, both sides will want to be able to move their units. Units are moved in response to orders—more about that later—but there are several things that must be checked before a unit can be moved around the battlefield:

- The program has to know what the maximum movement (number of squares) is for each unit. In Cavendish Field any troop unit's mobility depends solely on the weight of their armour, but in your games it could also depend on discipline, morale, exhaustion, and so on.
- Are there any advantages or bonuses to be had? (In this game, only the cavalry have bonuses, but you might give bonuses for charging, for travelling downhill, or having

an enthusiastic leader.)

- Are there any terrain hazards blocking the way? (It's up to you what effect terrain has, but here all terrain except plains reduces the distance a unit moves by one square.) Is another unit in the way?
- Has the edge of the map been reached?

Add this routine and the program will be able to test for these factors before moving any unit:

```
1160 REM Move unit
1170 LET ox=T(b,8): LET oy=T(b,9)
1175 LET z$="□"
1180 IF m(T(b,8),T(b,9))<>Ø THEN LET
   z$=CHR$ (143+m(T(b,8),T(b,9)))
1190 LET D=5-T(b,4)
1200 IF b<3 OR b=9 OR b=10 THEN LET
   D=D+2
1210 LET v=T(b,2)-1
1215 LET up=Ø: LET al=v-2
1220 IF v/2-(INT (v/2))=Ø THEN LET
   up=v-1: LET al=Ø
1230 REM Dummy for Repeat
1240 LET n1=T(b,9)+al: LET
   np=T(b,8)+up
1250 IF np<1 THEN LET np=1
1260 IF np>15 THEN LET np=15
1270 IF n1<1 THEN LET n1=1
1280 IF n1>30 THEN LET n1=30
1290 IF m(np,n1)>Ø THEN LET D=D-1
1300 FOR k=1 TO 8
1310 IF (T(k,9)=n1 AND T(k,8)=np AND
   k<>b) THEN LET D=Ø
1315 IF (T(k+8,9)=n1 AND T(k+8,8)=np
   AND k+8<>b) THEN LET D=Ø
1320 NEXT k
1330 IF D>Ø THEN LET T(b,9)=n1: LET
   T(b,8)=np: LET D=D-1
1340 IF D<>Ø THEN GOTO 1230
1350 INK Ø: PRINT AT ox,oy;z$
1360 INK cl: PRINT AT T(b,8),T(b,9);u$(i)
1370 RETURN
```

```
1160 REM MOVE UNIT
1170 OX=T(B,7):OY=T(B,8)
1175 GH=32
1180 IF M(T(B,7),T(B,8))<>Ø THEN
   GH=M(T(B,7),T(B,8))+63
1190 D=4-T(B,3)
1200 IF B<2 OR B=8 OR B=9
   THEND=D+2
1210 V=T(B,1)-1
1215 UP=Ø:AL=V-2
1220 IF V/2 -INT(V/2)=Ø THEN
   UP=V-1:AL=Ø
1230 REM DUMMY FOR REPEAT LOOP
1240 NL=T(B,7)+AL:NP=T(B,8)+UP
1250 IF NP<Ø THEN NP=Ø
1260 IF NP>16 THEN NP=16
1270 IF NL<Ø THEN NL=Ø
1280 IF NL>37 THEN NL=37
1290 IF M(NL,NP)>Ø THEN D=D-1
1300 FOR K=Ø TO 7
1310 IF(T(K,7)=NL AND T(K,8)=NP AND
   K<>B)THEN D=Ø
1315 IF(T(K+8,7)=NL AND T(K+8,8)=NP
   AND K+8<>B)THEN D=Ø
1320 NEXT K
1330 IF D>Ø THEN T(B,7)=NL:
   T(B,8)=NP:D=D-1
1340 IF D<>Ø THEN 1230
1350 CL=Ø:P=OX:Q=OY:
   GOSUB2600
1360 CL=CO:P=T(B,7):Q=T(B,8):
   GH=VAL(MID$(U$,B+1,1))+67:
   GOSUB2600
1370 RETURN
```

```
1160 DEF PROCmove(B%)
1170 ox=T%(B%,7)+1:oy=T%(B%,8)+1
1180 IF FNmread(T%(B%,7),T%(B%,8))<>Ø
   THEN oldter$=CHR$(223+ FNmread(T%
   (B%,7),T%(B%,8))) ELSE oldter$="□"
1190 D%=4-T%(B%,3)
1200 IF B%<2 THEN D%=D%+2
1210 dir=T%(B%,1)-1
1220 IF dir□DIV 2 =dir/2 THEN
   up=dir-1:al=Ø ELSE up=Ø:al=dir-2
1230 REPEAT
1240 nal=T%(B%,7)+al:nup=T%
   (B%,8)+up
1250 IF nup<Ø THEN nup=Ø
1260 IF nup>20 THEN nup=20
1270 IF nal<Ø THEN nal=Ø
1280 IF nal>37 THEN nal=37
1290 IF FNmread(nal,nup)>Ø THEN
   D%=D%-1
1300 FOR k=Ø TO 7
1310 IF (T%(k,7) = nal□AND T%(k,8) =
   nup□AND k<>B%) OR (T%(k+8,7) =
   nal□AND T%(k+8,8) = nup□AND
   k+8<>B%) THEN D% = Ø
1320 NEXT
1330 IF D%>Ø THEN T%(B%,7)=nal:
   T%(B%,8) =nup:D%=D%-1
1340 UNTIL D%=Ø
1350 COLOURØ:PRINT TAB(ox,oy);oldter$
1360 COLOURcl:PRINT TAB(T%
   (B%,7)+1,T%(B%,8)+1);
   MID$(unst$,(i□
   MOD 8)+1,1)
1370 ENDPROC
```

```
1160 REM MOVE UNIT
1170 OX=T(B,8):OY=T(B,9)
1175 ZZ=Ø
1180 IF M(T(B,8),T(B,9))<>Ø THEN
   ZZ=M(T(B,8),T(B,9))
1190 D=5-T(B,4)
1200 IF B<3 OR B=9 OR B=10 THEN
   D=D+2
1210 V=T(B,2)-1
1215 UP=Ø:AL=V-2
1220 IF (V/2)-INT(V/2)=Ø THEN
   UP=V-1:AL=Ø
1230 REM
1240 NL=T(B,9)+AL:NP=T(B,8)+UP
1250 IF NP<1 THEN NP=1
1260 IF NP>15 THEN NP=15
1270 IF NL<1 THEN NL=1
1280 IF NL>30 THEN NL=30
1290 IF M(NP,NL)>Ø THEN D=D-1
1300 FOR K=1 TO 8
1310 IF (T(K,9)=NL AND T(K,8)=NP AND
   K<>B) THEN D=Ø
1315 IF (T(K+8,9)=NL AND T(K+8,8)=NP
   AND K+8<>B) THEN D=Ø
1320 NEXT K
1330 IF D>Ø THEN T(B,9)=NL:
   T(B,8)=NP:D=D-1
1340 IF D<>Ø THEN 1230
1350 X9=OY*8:Y9=OX*8:IF ZZ<>Ø THEN
   COLOR 4:LINE(X9,Y9)-(X9+7,Y9+7),
   PRESET,BF:DRAW"BM"+STR$(X9)+
   ","+STR$(Y9)+UC$(ZZ) ELSE LINE
   (X9,Y9)-(X9+7,Y9+7),PRESET,BF
1360 COLOR CL:DRAW"BM"+STR$
   (T(B,9)*8)+","+STR$(T(B,8)*8):UU=
   VAL(MID$(U$,I,1)):A$=UC$(UU):
   GOSUB 3000
1370 RETURN
```

The tests will increase, reduce, or completely prevent movement. The routine first 'remembers' the old position, and the terrain of that position. It then calculates the direction and the maximum movement.

Lines 123Ø to 134Ø are a loop which tests each square along the unit's path to see if it is occupied by troops or terrain which would impede the unit's progress. According to what the loop finds, the distance the unit moves is adjusted.

The loop repeats until the distance variable is zero—the computer is back to the start square. A REPEAT ... UNTIL loop is the most appropriate way to check the path, but it's only available on the Acorn, so on the other machines a dummy REPEAT ... UNTIL loop has been set up. Several of these loops appear in the completed program, and are marked by a REM statement at the beginning of the loop.

Having decided on the range, the routine replaces the unit symbol with the original terrain. The unit is then displayed at its new position.

The next part of this article looks at how orders are issued to the units.

RECURSION-LOOPS WITHIN LOOPS

- AN AIR OF MYSTERY
- INTELLIGENT MACHINE
- RECURSIVE PROCEDURES
- PASSING PARAMETERS
- CHASING THE BUGS

If any of your programs have subroutines that are called repeatedly, then you might have a case for using recursive programming—find out how

Programming a computer is, essentially, an exercise in problem solving. As your programming skills develop, you should realize that most difficult problems can be solved if they are broken down into smaller, simpler problems. Eventually, however, you arrive at a programmer's nightmare in which an attempt to solve a problem leads only to a new problem, the solution of which leads to yet another problem, and so on.

You might consider this a sensible point to call a halt and go off in search of a different problem, but remember that your micro is not affected by the sort of mental barrier that would cause a human brain to seek the earliest opportunity to exit the nightmare. In fact, there is an advanced programming technique for solving certain types of problem that can be reduced to problems within problems. This technique is called recursion.

AN AIR OF MYSTERY

Generally, inexperienced programmers regard recursion as a highly complicated and mysterious topic. This is because the programming techniques involved can be extremely difficult to follow from the listing; sometimes even a flow chart does not show clearly what is actually going on. Nevertheless, the principle is not difficult to grasp.

In mathematical terms, 'recursion' is the repetition of a particular operation. This definition, however, is not strictly applicable in programming, where it has a more precise meaning. In essence, recursion is primarily a call to a sub-routine or procedure with an initial set of parameters. The subroutine or procedure then repeatedly calls itself, updating the parameters each time, until a particular task has been performed.

CLEVER MACHINE

To help you to understand the basic concept of recursion, think about how a driver might



make a journey from place A to place B across a large unfamiliar city. The driver has street maps of the city, but finds that travelling directly from A to B is too difficult. So he decides to break down the problem into a number of similar (but much simpler) problems, each of which can be solved in turn. The easiest way to do this is to select a location (C) lying between A and B and decide how to travel from A to C. He then drives to C.

Once at C, the driver looks to see whether he can travel directly from C to B. If this is possible, he does so; otherwise he repeats the above process by choosing a new location (D), between C and B. This process is repeated until he reaches the destination B.

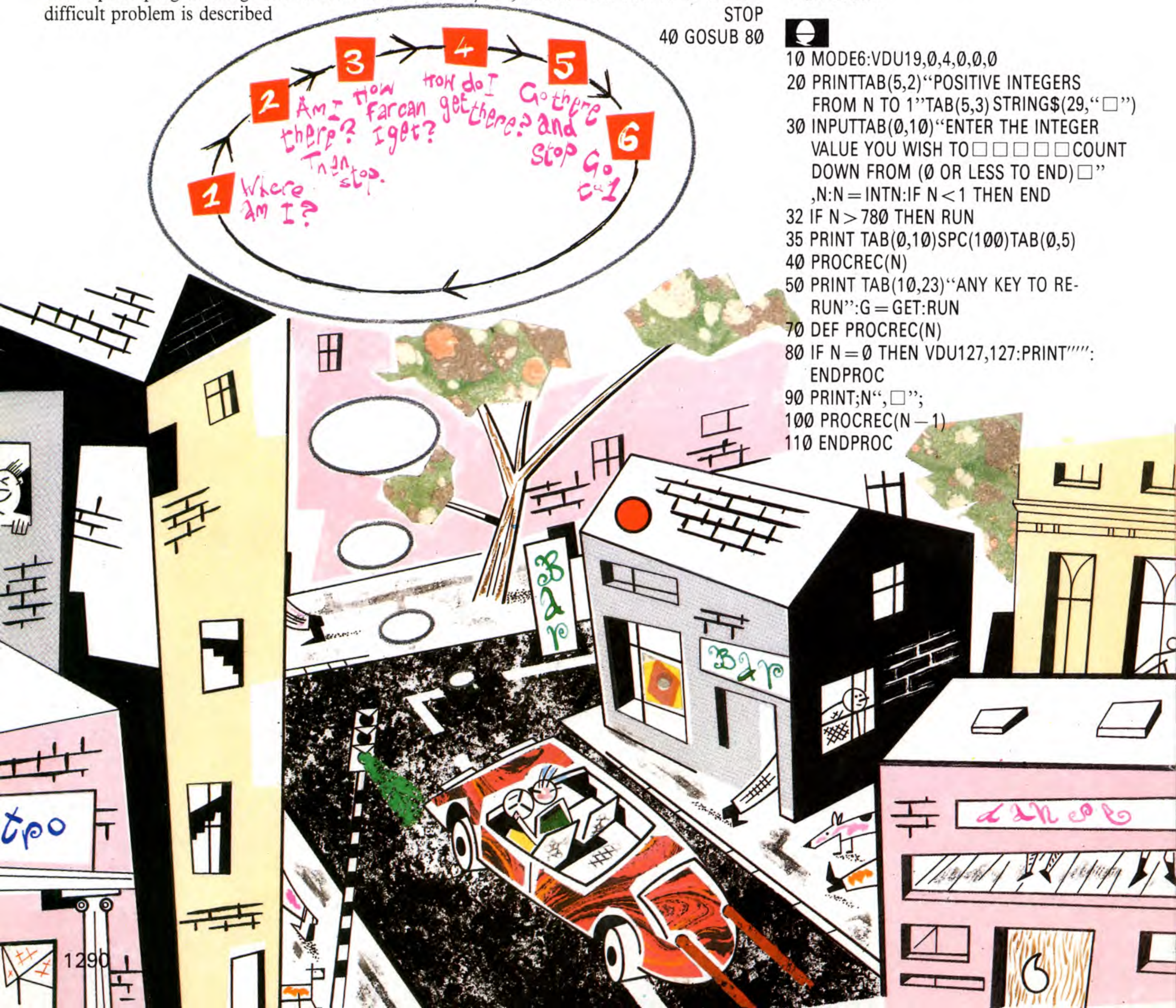
This simple example aptly demonstrates the principles of recursion as they are applied to computer programming. The solution to a difficult problem is described in terms of easier (or smaller) problems.

As each sub-task (or level) of recursion is reached, it is often necessary to store information concerning the previous position reached in the preceding level, to return successfully to it later. As each level is entered, a different set of parameters is issued and a test is made to determine when the entire task is completed. Without such a test, the process would never end. To see the method in use, enter and RUN the first program, which prints the positive integers from an input value (N) to one.

```
20 PRINT INVERSE 1;TAB 1;"□POSITIVE
   INTEGERS FROM N TO 1□"
30 INPUT "ENTER THE INTEGER YOU WISH
   TO□□□COUNT DOWN FROM (Ø TO
   END)□";N:LETN=INT N:IF N<1 THEN
                                STOP
40 GOSUB 8Ø
5Ø GOTO 3Ø
8Ø IF N=Ø THEN RETURN
9Ø PRINT ;N;"□□";
1ØØ LET N=N-1:GOSUB 8Ø
11Ø RETURN
```

```
2Ø PRINT "▣▣>▣▣POSITIVE
   INTEGERS FROM N TO 1"
3Ø PRINT "▣ENTER THE INTEGER VALUE
   YOU WISH TO":PRINT "COUNT DOWN
   FROM (1-22)"
35 INPUT N:N=INT(N):IF N<1 OR N>22
   THEN END
4Ø GOSUB 8Ø
5Ø GOTO 3Ø
8Ø IF N=Ø THEN RETURN
9Ø PRINT N",";
1ØØ N=N-1:GOSUB 8Ø
11Ø RETURN
```

```
1Ø MODE6:VDU19,Ø,4,Ø,Ø,Ø
2Ø PRINTTAB(5,2)"POSITIVE INTEGERS
   FROM N TO 1"TAB(5,3)STRING$(29,"□")
3Ø INPUTTAB(Ø,1Ø)"ENTER THE INTEGER
   VALUE YOU WISH TO□□□□□COUNT
   DOWN FROM (Ø OR LESS TO END)□"
   ,N:N=INTN:IF N<1 THEN END
32 IF N>78Ø THEN RUN
35 PRINT TAB(Ø,1Ø)SPC(1ØØ)TAB(Ø,5)
4Ø PROCREC(N)
5Ø PRINT TAB(1Ø,23)"ANY KEY TO RE-
   RUN":G=GET:RUN
7Ø DEF PROCREC(N)
8Ø IF N=Ø THEN VDU127,127:PRINT''''':
   ENDPROC
9Ø PRINT;N",□";
1ØØ PROCREC(N-1)
11Ø ENDPROC
```



```
10 CLS
20 PRINT"POSITIVE INTEGERS FROM N TO
   1"
30 PRINT:PRINT:INPUT"ENTER THE INTEGER
   VALUE YOU WISHTO COUNT DOWN FROM
   (Ø OR LESS TO END)□";N:N=INT(N):IF
   N<1 THENEND
40 GOSUB 80
50 GOTO 30
80 IF N=Ø THEN RETURN
90 PRINTN;",";
100 N=N-1:GOSUB80
110 RETURN
```

The program lets you input the value of the largest integer from which you wish to count down. Entering a value less than one stops the program. On the Commodores, values greater than 22 also stop the program, because these micros can remember only 23 jumps to a subroutine. Line 4Ø calls the recursive subroutine, the first line of which tests for the completion of the entire task.

This test is crucial for ending the recursive calls. The first level of recursion is entered with N as specified by you. This value is printed at Line 9Ø. The second level is entered at Line 1ØØ, which reduces the value of N by one and calls the subroutine again with the new value of N. The program continues to loop between Lines 8Ø and 1ØØ, printing each integer in turn.

When N is reduced to Ø (at Line 1ØØ), the program branches as usual to Line 8Ø, where this time it must obey the RETURN. This causes a return from the subroutine called at Line 1ØØ. The next instruction is at Line 11Ø, which causes a return from the subroutine called at Line 4Ø. The next instruction (Line 5Ø) effectivley runs the program again.

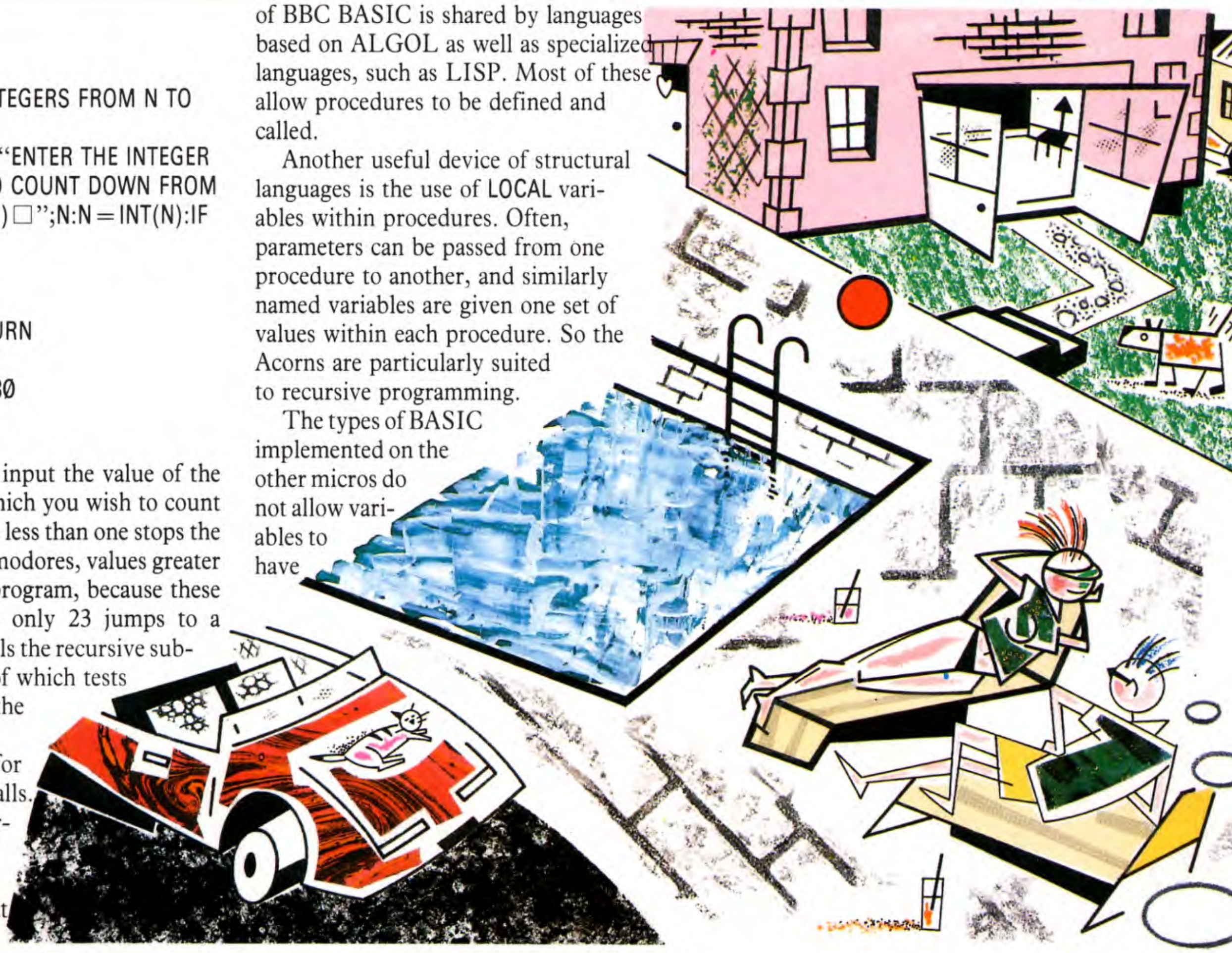
Notice that the program ends with a value of N=Ø, set at Line 1ØØ, but Line 9Ø never prints this value. To reset N to the last value printed, you could enter N=N+1 (LET N=N+1 for the Spectrum) at Line 1Ø5. Then N will be set to the last value printed.

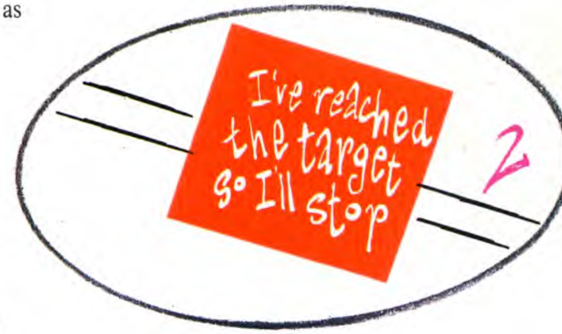
RECURSIVE PROCEDURES

On the Acorns, the use of PROCedures makes it easy to pass parameters to subroutines. So at Line 1ØØ, for example, a single statement decrements N by one and calls the recursive subroutine. Using the other micros' versions of BASIC, two statements are required. However, on these machines, the structured form of BBC BASIC is shared by languages based on ALGOL as well as specialized languages, such as LISP. Most of these allow procedures to be defined and called.

Another useful device of structural languages is the use of LOCAL variables within procedures. Often, parameters can be passed from one procedure to another, and similarly named variables are given one set of values within each procedure. So the Acorns are particularly suited to recursive programming.

The types of BASIC implemented on the other micros do not allow variables to have more than one value but, fortunately, they do allow GOSUBs to call themselves—such as PROCedures do. Programming recursive calls on these machines, therefore, is only slightly more complicated than in BBC BASIC. Enter the second program to see a simple demonstration of how the problem of variables is solved.






```
10 DIM N(34):DIM A(34)
20 CLS
30 PRINT TAB 3;INVERSE 1;"CALCULATION
   OF FACTORIALS"
40 INPUT "ENTER THE FACTORIAL NUMBER
   YOU□□REQUIRE (1-33, OR Ø TO
   END)";NU
50 IF NU>33 OR NU<>INT (NU) OR
   NU<Ø THEN RUN
60 IF NU=Ø THEN STOP
70 LET LE=1:LET N(LE)=NU:LET AN=NU
80 GOSUB 150
90 PRINT AN;"!□=□";A(1):PRINT:GOTO 40
150 IF N(LE)=Ø THEN LET A(LE)=1:GOTO
   180
160 LET LE=LE+1: LET
   N(LE)=N(LE-1)-1: GOSUB 150
170 LET LE=LE-1: LET
   A(LE)=A(LE+1)*N(LE)
180 RETURN
```

```
10 DIM N(34),A(34)
30 PRINT "■■>■■CALCULATION
   OF":PRINT"■■■□□FACTORIALS
   □□■"
40 PRINT "ENTER THE FACTORIAL NUMBER
   YOU REQUIRE"
45 PRINT "(1-33, OR Ø TO END)":INPUT NU
50 IF NU>33 OR NU<>INT(NU) OR
   NU<Ø THEN RUN
60 IF NU=Ø THEN PRINT "■":END
```

```
70 LE=1:N(LE)=NU:AN=NU
80 GOSUB 150
90 PRINT AN"!▟ = ▟";A(1)"▙":GOTO
   40
150 IF N(LE)=0 THEN A(LE)=1:GOTO 180
160 LE=LE+1:N(LE)=N(LE-1)-1:
   GOSUB 150
170 LE=LE-1:A(LE)=A(LE+1)*N(LE)
180 RETURN
```

```
20 MODE1:VDU19,0,4,0,0,0
30 PRINT TAB(7,2)"CALCULATION OF
   FACTORIALS:"TAB(7,3)STRING$(26,"□")
40 INPUT" TAB(0,13)"ENTER THE
   FACTORIAL NUMBER (1-33, OR 0 TO
   END)□",NU
50 IF NU>33 OR NU<>INT(NU) OR
   NU<0 THEN 20
60 IF NU=0 THEN END
70 AN=1
80 PROCREC(1)
90 PRINT TAB(13,20);NU;"!□EQUALS□";
   AN:PRINT TAB(10,25)"ANY KEY TO
   RE-RUN":G=GET:RUN
140 DEF PROCREC(T)
150 IF T<>NU THEN AN=AN*(T+1):
   PROCREC(T+1)
180 ENDPROC
```

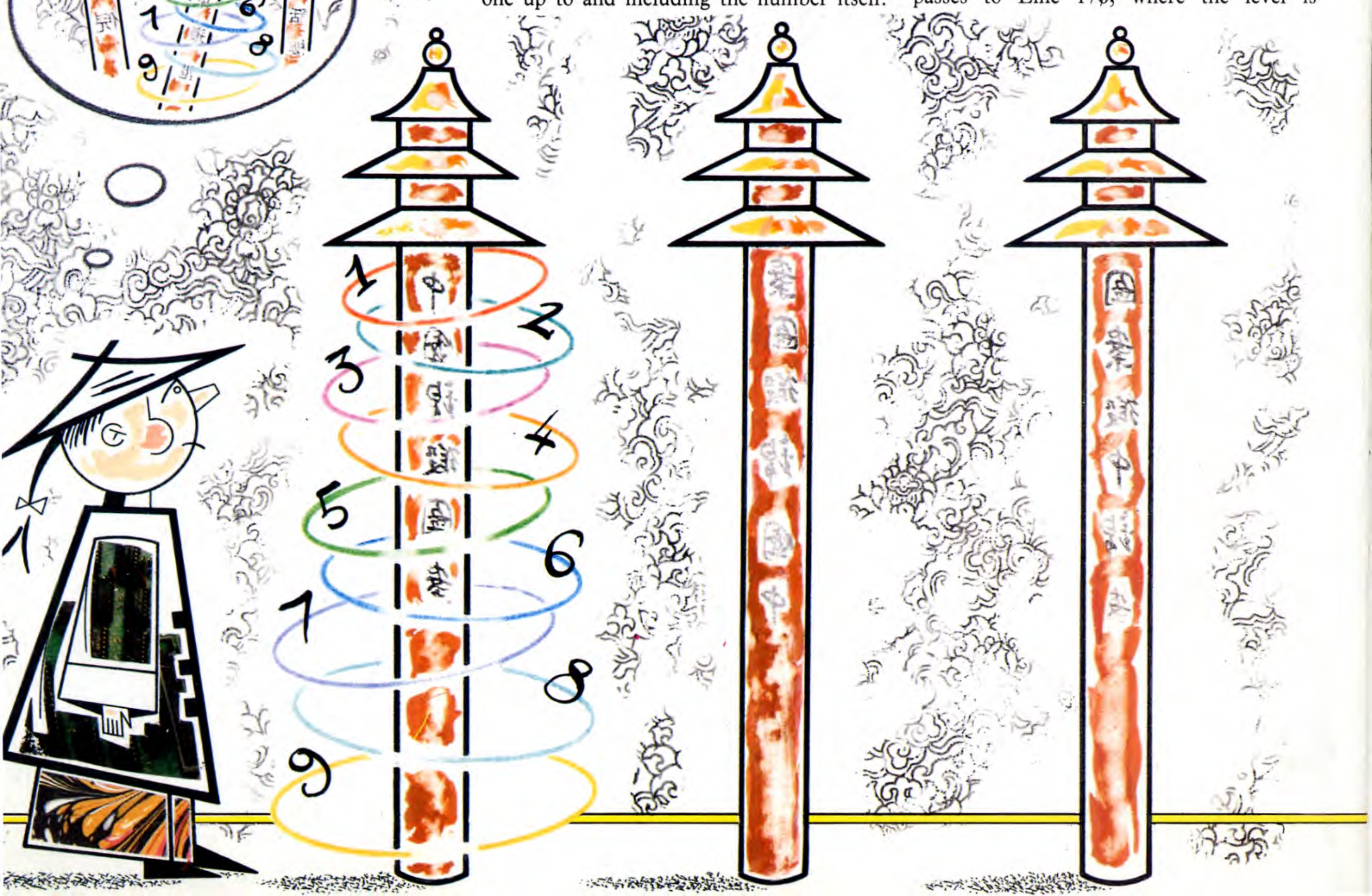
```
10 DIM N(34),A(34)
20 CLS
30 PRINT@3,"calculation of factorials"
40 INPUT"ENTER THE FACTORIAL NUMBER
   YOU□□REQUIRE (1-33, OR 0 TO
   END)";NU
50 IF NU>33 OR NU<>INT(NU) OR
   NU<0 THEN20
60 IF NU=0 THEN END
70 LE=1:N(LE)=NU:AN=NU
80 GOSUB150
90 PRINTAN"!□=□";A(1):PRINT:
   GOTO40
150 IF N(LE)=0 THEN A(LE)=1:GOTO 180
160 LE=LE+1:N(LE)=N(LE-1)-
   1:GOSUB150
170 LE=LE-1:A(LE)=A(LE+1)*N(LE)
180 RETURN
```

RUN the program and enter a value in response to the prompt. The program calculates and prints the *factorial* of the number you entered—the product of every integer from one up to and including the number itself. For example, factorial 5 (written 5!) is 1 × 2 × 3 × 4 × 5 or 120. Factorial calculations are frequently required for certain statistical applications, so a simple method of generating them is useful. But the best method depends on the language you are using.

For micros other than the Acorns, the program DIMensions variables (Line 10) in sufficient numbers to complete the task. These array variables (see page 152) reserve memory space for their use only, and use the recursion level (LE) as the subscript of the variable currently in use—N(LE).

The program essentially begins at Line 70, where the level is set to one. Here also the number you input—five, say—is set to N(1) and to AN (the variable that accumulates the answer). Line 80 then calls the first level of recursion. The first line of the recursion routine (Line 150) tests for the end of the problem—when N(LE)=0. But at the moment N(LE) is five, so control passes to Line 160. This increments the level (to two), sets the current number to 4, then calls the recursive routine again, and so on. When Line 160 increments the level to six and decrements the current number to zero, Line 150 detects that N(6) is zero, so element six of the A array is set to one, and Line 180 issues a RETURN to the end of Line 160. Control now passes to Line 170, where the level is





decremented to five and A(5) is set to A(6) times N(5). This makes A(5) equal 1 × 1 or 1. Line 18Ø now returns control to the end of Line 15Ø again, where this time A(4) is set to A(5) times N(4). So A(4) equals one (calculated above) times 2. This loop is continued for as many times as the GOSUB at Line 16Ø was called. When the loop is completed, LE equals one, and the last RETURN is to Line 8Ø. The next instruction (Line 9Ø) prints the result—12Ø.

PASSING PARAMETERS

By passing parameters within procedures, the Acorn program avoids the need to DIMension arrays, and there is no confusion with variables either. Line 15Ø merely checks whether the number of recursive levels equals the number you have entered. If it doesn't, each calculated value is accumulated in AN, then the procedure is called for successive levels. When the last level is reached, Line 18Ø returns control to Line 9Ø, which prints the answer.

There are in practice two limitations to the application of recursion on computers—even those, like the Acorns, that have structured BASIC. The first is that there are limits to the value of variables that can be handled. This fact must be taken into account in any sort of programming, but it is even more important in recursive routines.

The second limitation is to do with the number of times a subroutine or procedure can call itself. This affects the number of levels of recursion that you can allow. Each call requires the system to remember where it left off before the call was made. This is achieved by placing pointers on the stack. Space is needed to store the values of the variables at each level. Eventually, the memory is used up, the stack becomes corrupted and the program crashes. However, with this program the limit of 33 levels is due to the fact that factorial 34 exceeds $1.7 \times 1Ø^{38}$, the maximum number which the computer can hold.

CHASING THE BUGS

To keep your program within the limits of the micro and prevent crashing, you must know how it behaves at the lowest level of recursion. Usually, the only way to ensure it is working correctly is to use test data of which you are certain. A simple method is to consider a recursive subroutine as a number or similar copies of the same subroutine. Note that to make conditional jumps out of subroutines is usually considered bad programming practice, because the stack is left undefined.

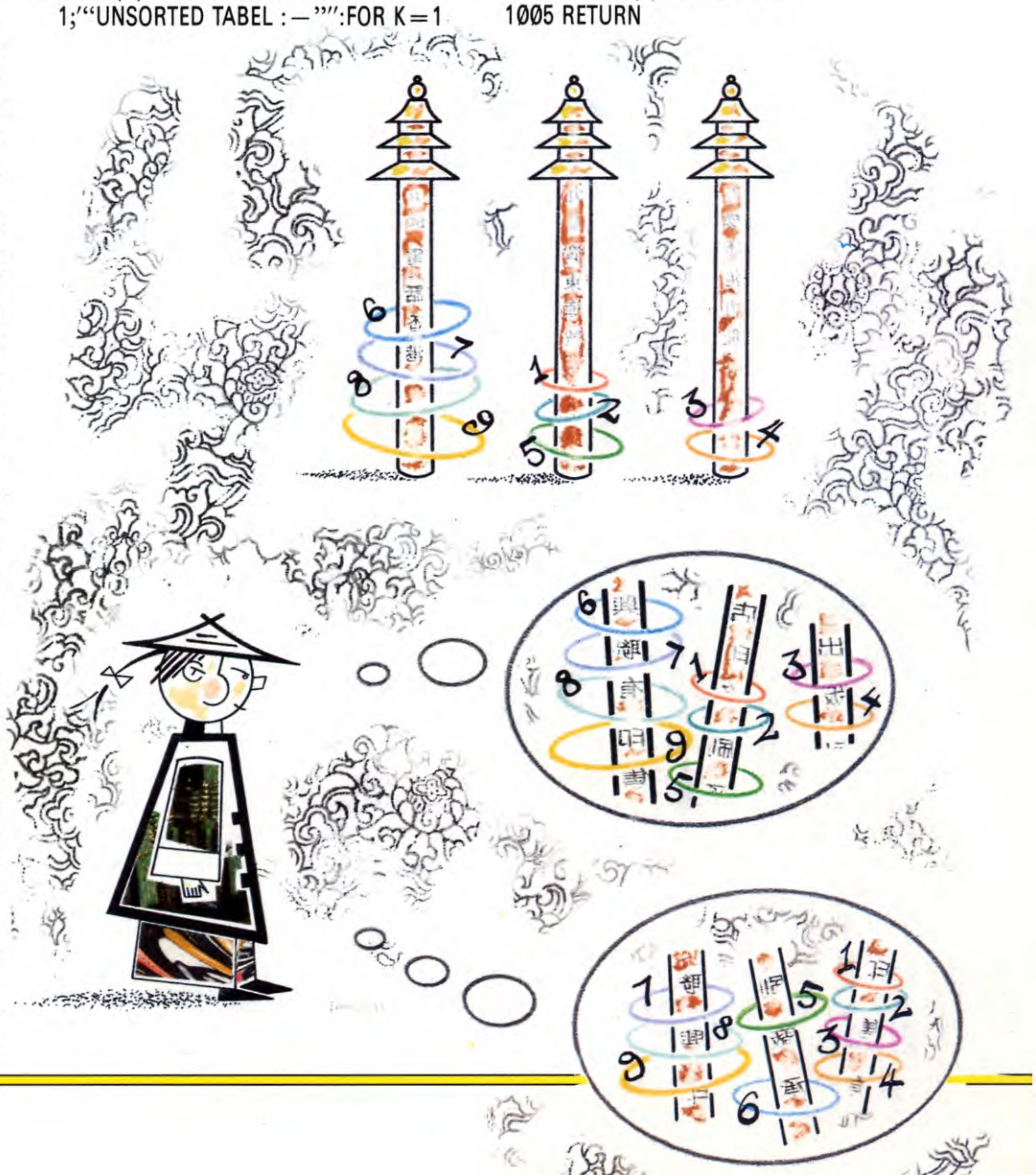
When you are writing this kind of subroutine, always start with an effective-exit test. This test is used to decide when the problem (as set by the input parameters) can be solved directly, without the need for sub-division.

Before you begin programming, plan what you want the subroutine to achieve. Then, when you are writing the program, don't worry too much about the exact sequence of execution, but consider the two main principles—the stopping condition and the sub-division into easier problems. By following this method, you should be able to program some of the more practical uses of recursion. Here is one example of how recursion can greatly improve the clarity and efficiency of a sort program.



The Dragon displays a nine-disc selection

```
1Ø BORDER 1:INK 7:PAPER 1:CLS
2Ø PRINT TAB 11;INVERSE
   1;"□QUICKSORT□"
3Ø INPUT "HOW MANY NUMBERS DO YOU
   WISH TO SORT (1-1ØØØ)□";A
4Ø IF A<1 OR A>1ØØØ THEN GOTO 1Ø
5Ø DIM A(A):DIM R(2+SQR(A))
6Ø LET A(A)=1ØØ:PRINT INVERSE
   1;'"UNSORTED TABEL :-"'':FOR K=1
   TO A-1:LET A(K)=INT (RND*99):PRINT
   A(K);"□";:NEXT K
7Ø LET L=1:LET LV=1:LET
   R=A-1:GOSUB 1ØØØ
8Ø PRINT INVERSE 1; '"SORTED TABLE
   :-"'':FOR K=1 TO A-1:PRINT
   A(K);"□";:NEXT K
9Ø IF INKEY$<>"□" THEN GOTO 9Ø
1ØØ RUN
1ØØØ IF R>L THEN LET I=L: LET J=R+1:
   LET V=A(L): GOTO 1Ø1Ø
1ØØ5 RETURN
```



```
1010 LET I=I+1:IF A(I)<V THEN GOTO
  1010
1020 LET J=J-1:IF A(J)>V THEN GOTO
  1020
1030 IF J>=I THEN LET T=A(I): LET
  A(I)=A(J):LET A(J)=T:GOTO 1010
1040 LET T=A(L):LET A(L)=A(J):LET
  A(J)=T
1050 LET R(LV)=R:LET LV=LV+1:LET
  R=J-1:GOSUB 1000
1060 LET LV=LV-1:LET R=R(LV): LET
  L=1:GOSUB 1000
1070 RETURN
```

```
10 PRINT "■■>■■QUICKSORT"
30 PRINT "■HOW MANY NUMBERS DO
  YOU WISH TO SORT":INPUT
  "(1-300)■";A
40 IF A<1 OR A>300 THEN RUN
50 DIM A(A),R(1+SQR(A))
60 A(A)=100:PRINT "■■■UNSORTED
  TABLE :-":FOR K=0 TO
  A-1:A(K)=INT(RND(1)*99)
65 PRINT A(K);:NEXT K:PRINT
70 L=0:R=A-1:GOSUB 1000
80 PRINT "■■■SORTED TABLE :-":FOR
  K=0 TO A-1:PRINT A(K);:NEXT K
90 GET Z$:IF Z$<>"□" THEN 90:
100 RUN
1000 IF R>L THEN I=L:
  J=R+1:V=A(L):GOTO 1010
1005 RETURN
1010 I=I+1:IF A(I)<V THEN 1010
1020 J=J-1: IF A(J)>V THEN 1020
1030 IF J>=I THEN T=A(I):A(I)=A(J):
  A(J)=T:GOTO 1010
1040 T=A(L):A(L)=A(J):A(J)=T
1050 R(LV)=R:LV=LV+1:R=J-
  1:GOSUB 1000
1060 LV=LV-1:R=R(LV):L=I:GOSUB
  1000
1070 RETURN
```

```
10 MODE1
20 PRINT TAB(15,3)"QUICKSORT"
30 INPUT TAB(7,5)"HOW MANY NUMBERS
  (1-750)□",A
40 A=INT A:IF A<1 OR A>750 THEN 10
50 DIM A(A)
60 A(A)=100:PRINT'"UNSORTED
  TABLE:-":FOR K=0 TO
  A-1:A(K)=RND(99):PRINT A(K);:NEXT
70 PROCSORT(0,0,A-1)
80 PRINT'"SORTED TABLE:-":FOR K=0
  TO A-1:PRINTA(K);:NEXT
90 PRINT'"PRESS SPACE BAR TO RE-
  RUN":REPEAT UNTIL GET=32:RUN
100 DEF PROCSORT(LV,L,R)
1000 IF R>L THEN I=L:J=R+1:V=A(L)
  ELSE ENDPROC
1010 I=I+1:IF A(I)<V THEN 1010
1020 J=J-1:IF A(J)>V THEN 1020
1030 IF J>=I□THEN T=A(I):A(I)=A(J):
  A(J)=T:GOTO 1010
1040 T=A(L):A(L)=A(J):A(J)=T
1050 PROCSORT(LV+1,L,J-1)
1060 PROCSORT(LV,I,R)
1070 ENDPROC
```

```
10 CLS
20 PRINT@11,"quicksort"
30 PRINT:INPUT"□HOW MANY NUMBERS
  DO YOU WISH TO SORT (1-1000)□";A
40 IF A<1 OR A>1000 THEN 10
50 DIM A(A),R(1+SQR(A))
60 A(A)=100:PRINT"□UNSORTED
  TABLE□:-":FORK=0TOA-1:A(K)=
  RND(99):PRINTA(K);:NEXT:PRINT
70 L=0:R=A-1:GOSUB1000
80 PRINT" SORTED TABLE□:-":FORK=
  0TOA-1:PRINTA(K);:NEXT
90 IF INKEY$<>"□" THEN 90:ELSE RUN
1000 IF R>L THEN I=L:J=R+1:V=A(L)
  ELSERETURN
1010 I=I+1:IF A(I)<V THEN1010
1020 J=J-1:IF A(J)>V THEN1020
1030 IF J>=I THEN T=A(I):A(I)=A(J):
  A(J)=T:GOTO1010
1040 T=A(L):A(L)=A(J):A(J)=T
1050 R(LV)=R:LV=LV+1:R=J-1:GOSUB
  1000
1060 LV=LV-1:R=R(LV):L=I:GOSUB
  1000
1070 RETURN
```

Compare this listing of a recursive Quicksort program with the non-recursive Quicksort program on page 711. The recursive listing is far less cluttered with variables and IF . . . THEN . . . GOTO . . . conditions, so it is much simpler to follow.

The program lets you enter a value for the number of random numbers you wish to sort. Line 50 DIMensions an array—A(A)—to store these numbers, and an R array to store variables for the recursive calls. This second array is not necessary on the Acorns. Line 60 generates and prints the unsorted random numbers, and Line 70 calls the recursive subroutine to sort them into ascending order. Line 80 prints the table of numbers.

The method uses some aspects of list merging from two subsets and some aspects of sub-list sorting. The main list is divided into two subsets (Lines 1010 and 1020). Notice the crucial exit test (Line 1000) to determine when recursion should end. Each of the two subsets is then sorted in one pass (Line 1030). The subsets are further divided, but two are merged into one of the new subsets, then each of the new subsets is sorted in the next pass. The subroutine then calls itself (Line 1050) repeatedly to complete the sort.

Although this BASIC sorting method is not as fast as machine code, it is extremely rapid. For example, to sort 100 values on the Spectrum takes about 40 seconds; a similar bubblesort routine would take well over one hour.

EXTENDED RANGE

The usefulness of recursion goes far beyond the calcualtion of factorials and other mathematical applications. Recursion can be extremely useful in games programs and for producing complicated graphical patterns. The technique can also be applied to artificial intelligence (AI), for example, both in robotic control and in games programs. Similar methods are also being used in language processing (compilers and interpreters).

Recursion is used in chess and strategy-game programs. Enter the next program to see a simple illustration of this use on the classic puzzle, the Towers of Hanoi.

```
10 BORDER 6:PAPER 6:INK 0:CLS
20 PRINT TAB 8;INVERSE 1;"□TOWERS OF
  HANOI□"
30 INPUT "ENTER NUMBER OF RINGS
  (2-9)□";N:IF N<2 OR N>9 THEN
  GOTO 30
35 DIM T(3)
36 LET A$="■□■":INK 2:FOR M=21
  TO 21-N STEP -1:PRINT AT M,7;A$;AT
  M,15;A$;AT M,23;A$:NEXT M:INK 0
37 PRINT INK 2;AT 21,7;"■■■";AT
  21,15;"■■■";AT 21,23;"■■■"
38 FOR M=1 TO N:PRINT INK 7;PAPER 0;AT
  20-T(1),8;N+1-M:LET
  T(1)=T(1)+1:NEXT M
39 PRINT #1;AT 0,9;"ANY KEY TO START"
40 PAUSE 0: PRINT #1;AT 0,8;"□□□□
  □□□□□□□□□□□□□□"
45 LET TT=2:LET TF=1:LET R=3
50 GOSUB 90
70 PRINT AT 10,8;"TOTAL MOVES
  =□";2↑N-1
80 STOP
90 IF N=0 THEN RETURN
100 LET N=N-1:LET W=R:LET R=TT:LET
  TT=W:GOSUB 90:LET W=R:LET
  R=TT:LET TT=W
110 GOSUB 200
120 LET W=R:LET R=TF:LET
  TF=W:GOSUB 90:LET W=R:LET
  R=TF:LET TF=W
130 LET N=N+1:RETURN
```

```
200 PRINT AT 20-(T(TF)-1),TF*8;"□";
    INVERSE 1;AT 20-(T(TT)),TT*8;N+1:LET
    T(TF)=T(TF)-1:LET T(TT)=T(TT)+1
210 BEEP .01,TT*T(TT)*2
220 RETURN
```

```
20 PRINT "■■>■■TOWER OR
   HANOI"
30 PRINT "■NUMBER OF RINGS (2-9)"
40 INPUT N:IF N<2 OR N>9 THEN RUN
50 TT=2:TF=1:R=3:GOSUB 90
60 PRINT "■MOVES TAKEN=";2↑N-1
70 END
90 IF N=0 THEN RETURN
100 N=N-1:W=R:R=TT:TT=W:GOSUB
    90:W=R:R=TT:TT=W
110 PRINT "RING";N;"FROM TOWER":PRINT
    TF;"TO TOWER";TT
120 W=R:R=TF:TF=W:GOSUB 90:W=R:
    R=TF:TF=W
130 N=N+1:RETURN
```

```
10 DIML(3)
20 MODE1:VDU19,0,4,0,0,0,23;8202;0;0;0;
30 PRINTTAB(13,2)"TOWERS OF HANOI"TAB
   (13,3)STRING$(15,"□")
40 PRINTTAB(9,5)"NUMBER OF RINGS
   (2-9)□";
50 N=GET-48:IF N<2 OR N>9 THEN 50
   ELSE PRINT;N
60 L(1)=N:L(2)=0:L(3)=0:FOR T=1 TO
   N:PRINTTAB(13,25-N+T);T:NEXT:T=0
70 PRINTTAB(7,9)"NUMBER OF MOVES
   TAKEN□="
80 PROCREC(N,3,1,2)
90 END
100 DEF PROCREC(N,TT,TF,R)
110 IF N=0 THEN ENDPROC
120 PROCREC(N-1,R,TF,TT)
130 L(TF)=L(TF)-1:L(TT)=L(TT)+1
140 PRINTTAB(TT*6+7,26-L(TT));N
150 SOUND17,-15,28*L(TT),1
160 T=T+1:PRINTTAB(32,9);T
170 PRINTTAB(TF*6+7,25-L(TF))"□"
180 PROCREC(N-1,TT,R,TF)
190 ENDPROC
```

```
10 CLS:DIMH(3)
20 PRINT@8,"towers of hanoi":PRINT
30 PRINT"NUMBER OF RINGS (2-9) ?";
40 A$=INKEY$:IF A$<"2" OR A$>
   "9" THEN 40
50 N=VAL(A$):H(0)=N:PRINT@64
60 FORK=0TO8:FORJ=0TO2:PRINT
   @165+K*32+J*9,CHR$(175)+
   "□"+CHR$(175);:NEXTJ,K
70 FORK=0TO2:PRINT@453+K*9,
   STRING$ (3,175);:NEXT
80FORK=1TON:POKE1478-32*K,
   49+N-K:NEXT
90 TT=1:TF=0: R=2
100 GOSUB1000
110 PRINT@65,"NUMBER OF MOVES
    TAKEN=";INT(2↑N-1)
120 PRINT"□PRESS A KEY TO RUN AGAIN"
130 IF INKEY$="" THEN 130
140 RUN
1000 IF N=0 THEN RETURN
1010 N=N-1:W=R:R=TT:TT=W:GOSUB
     1000:W=R:R=TT:TT=W
1020 POKE1478+9*TF-32*H(TF),
     96:H(TF)=H(TF)-1:H(TT)=H(TT)
     +1:POKE1478+9*TT-32*H(TT),49+N
1030 W=R:R=TF:TF=W:GOSUB
     1000:W=R:R=TF:TF=W
1040 N=N+1:RETURN
```

The traditional puzzle consists of three pegs mounted on a board. On the first peg are a number of discs of varying diameter. The object is to transfer the stack of discs from one peg to another. The discs may be moved only one at a time, and no disc can ever rest on another smaller than itself. The third peg is used as a temporary rest while the discs are being moved. The computer version gives a graphic representation of the puzzle. RUN the program and enter the number of discs you wish to demonstrate. The transfer of discs is rapid so you won't be able to follow each move, unless you modify the program to slow it down, as follows.

For the Spectrum, insert a new line:

```
215 PAUSE 0.
```

For the Commodores, enter a new line:

```
115 POKE 198,0:WAIT 198,1
```

For the Acorns, enter two new lines:

```
65 G=GET
175 G=GET
```

The above changes cause the programs to run only when you press a key.

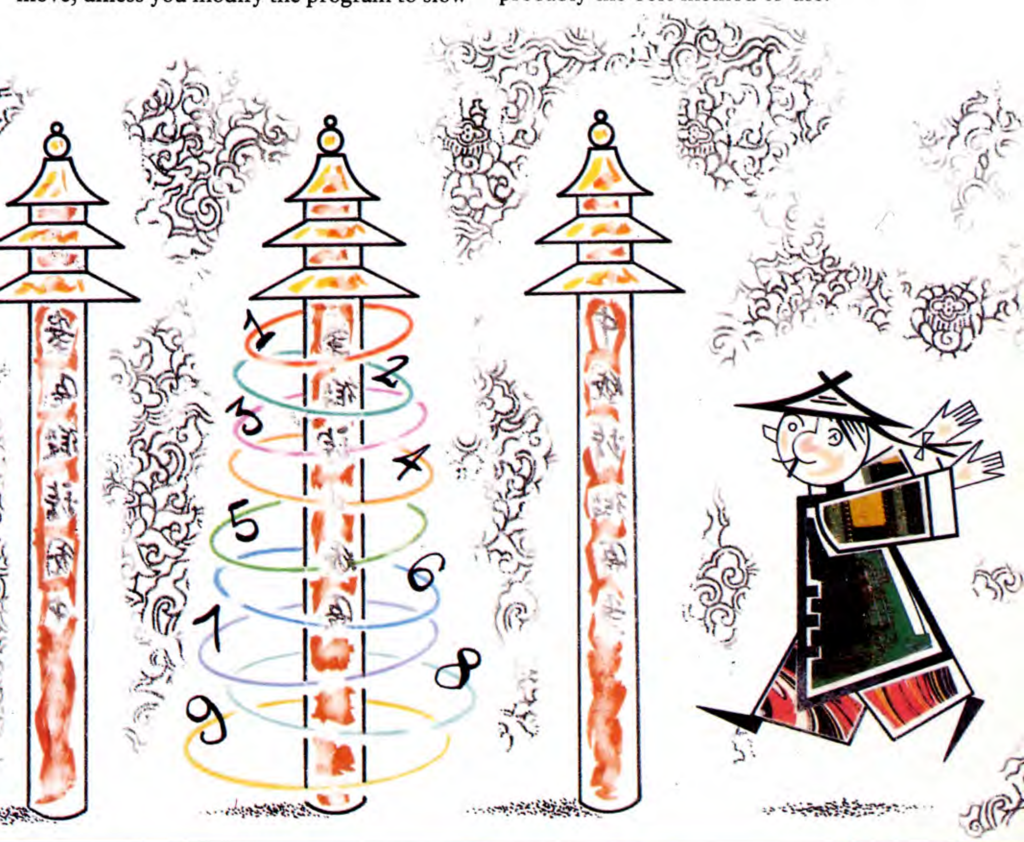
For the Dragon and Tandy, add the following to the end on Line 1020

```
: SOUND 50+H(TT)*10, 12
```

This causes a delay while a note is sounded.

The program to solve the puzzle is best coded for recursion. The format is similar to that of the previous programs. The recursive subroutine successively prints each move from one peg to another until the problem is completed. The total number of moves made is calculated and displayed after each full go.

Hence, for highly complicated problems which require sub-division, recursion is often the best method of programming. If, however, memory space is limited, then recursion can be extremely wasteful in both time and space. But if a subroutine or procedure calls itself more than twice, recursion is most probably the best method to use.



TURNING TURTLE WITH LOGO

Stars, circles, spirals and hexagons, magically drawn in ever increasing sizes—these are only a few of the shapes you can conjure up with a few simple instructions using LOGO

In the first part of this article you were shown how to draw with LOGO's Turtle, create a Logo primitive by teaching the Turtle to draw a picture, and how to use those primitives to help define new primitives. For example, it is possible to teach the Turtle to draw a hexagon:

```
TO HEXAGON
REPEAT 6 [FORWARD 70 RIGHT 60]
END
```

After typing END, LOGO will indicate HEXAGON DEFINED letting you know that HEXAGON is now part of its vocabulary. When you type HEXAGON, the Turtle will draw a hexagon and finish facing in the same direction as it started.

HEXAGON can be used to define PATTERN:

```
TO PATTERN
REPEAT 12 [HEXAGON FORWARD 10 RIGHT
30]
END
```

It is unusual if your LOGO procedures do what you expect the first time. There is usually at least one bug lurking somewhere in the program. For example, when drawing the hexagon you may have assumed that because the angle between the sides is 120 degrees, you should tell the Turtle REPEAT 6 [FORWARD 60 RIGHT 120]. The result would not have been a hexagon.

THE TURTLE'S EYE VIEW

Turtle geometry is unlike the coordinate geometry with which you are probably familiar, where positions are defined in relation to an external point. (Normal screen coordinates on your computer's graphics screen move like this.) Turtle geometry works differently as the Turtle itself is the point of reference. When drawing a hexagon the Turtle turns through 60 degrees, not 120 degrees, even though the two lines the Turtle has drawn are at 120 degrees to each other. If you have problems converting to Turtle geometry, imagine you are walking through the shape you want to draw. The angles you turn through and the distances you walk will translate into instructions for the Turtle.

Although this may seem confusing when you are used to coordinate geometry, it is actually far easier for children to learn, as it relates to their own experience.

EDITING

When the bugs have wreaked their havoc they need to be removed from the program. To do this you need to go into a third mode, the Editor. The Edit mode is similar to the procedural mode. It does not affect the immediate state of the Turtle.

To edit a procedure type EDIT " and the procedure's name. Some versions of LOGO do not require the quotation marks. EDIT can be abbreviated to ED.

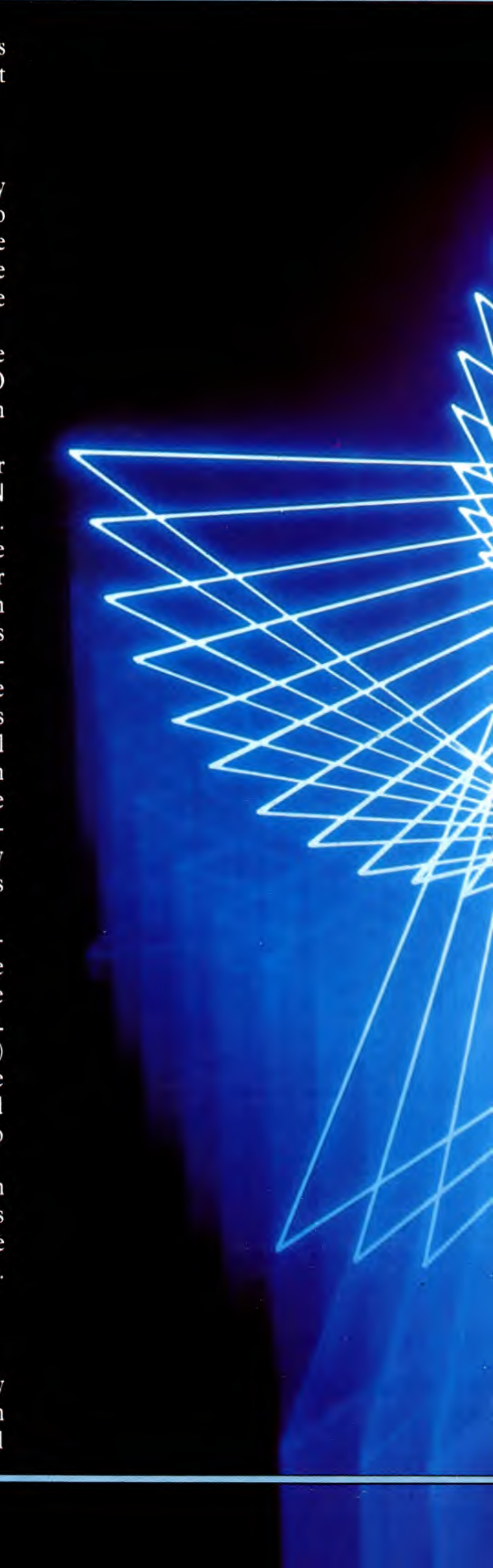
Type EDIT "HEXAGON and you will enter the EDIT mode. The definition of HEXAGON will be displayed at the top of the screen. There are no graphics in the Edit mode, the screen is devoted to text. Using the cursor keys, the cursor can be moved up and down the program from line to line, or backwards and forwards across an individual line. Individual characters to the left of the cursor can be removed with the delete key, the same way as an individual line is edited in the procedural mode. The character beneath the cursor can be removed with the delete key and the Control key (or CAPS SHIFT on the Spectrum). Characters are inserted by simply typing them in, the rest of the line moves along to accommodate them.

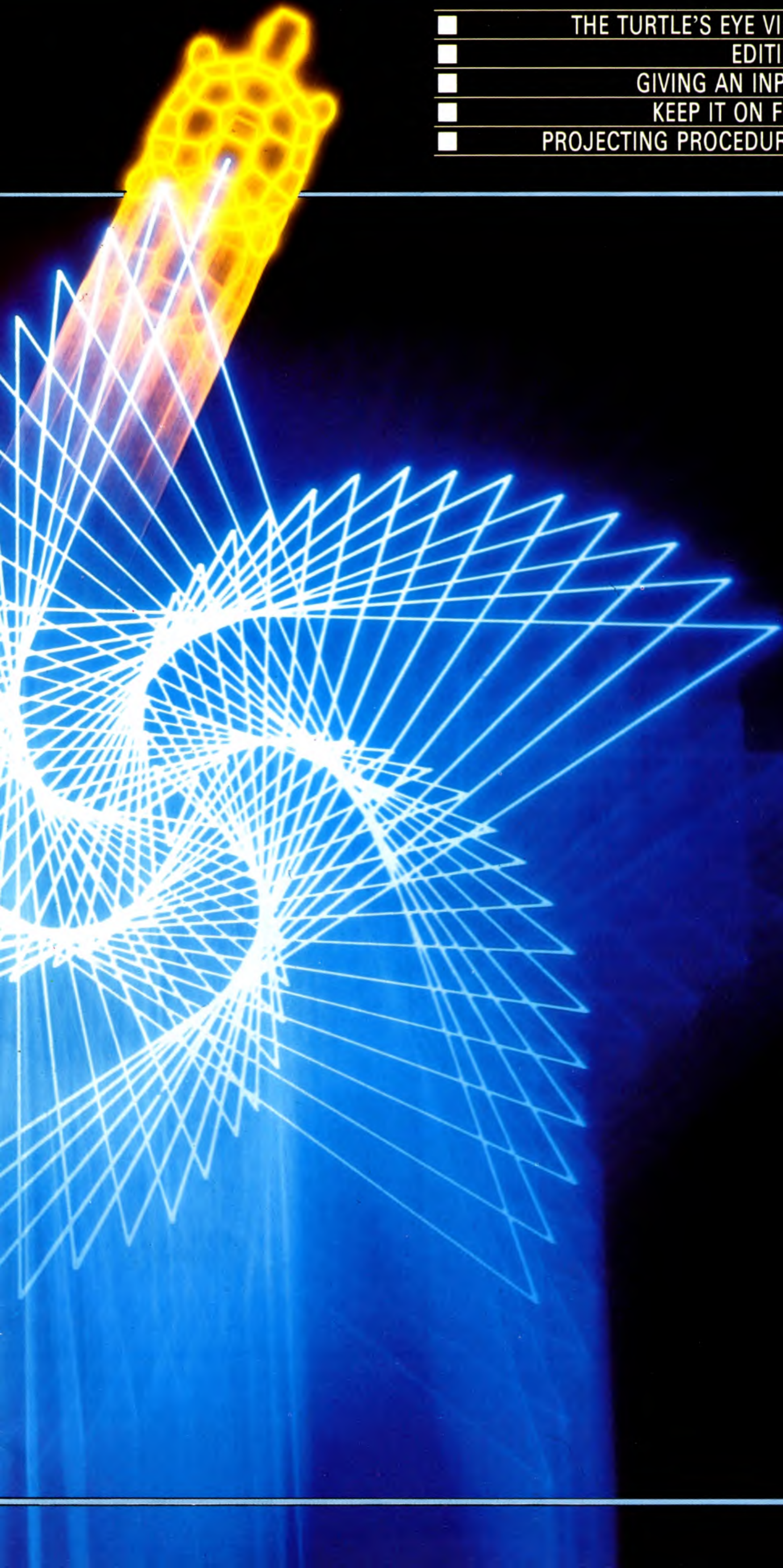
The procedures are the same on the machines, though the keys may differ. On the Commodore, for example, CTRL A moves the cursor to the beginning of a line, CTRL L moves it to the end of a line. CTRL K (for Kill) deletes all the characters to the right of the cursor. To delete a whole line use CTRL A and then CTRL K. CTRL O will open a space to insert a new line.

It is possible to define a new procedure in the Edit mode. The advantage of doing this is that you can move from line to line to change commands while still defining the procedure. To define Flower in this way type

```
EDIT "FLOWER
```

TO FLOWER will appear at the top of an empty screen. You can then write the procedure in the normal way and enjoy the benefit of full





editing capabilities.

When defining a procedure in Commodore LOGO you go into the editing mode whether you type TO or EDIT.

GIVING AN INPUT

There is an important difference between HEXAGON and RIGHT. HEXAGON always draws a hexagon with sides 60 units. RIGHT requires an input which tells the Turtle how far RIGHT it must turn, in the same way FORWARD, BACK, and LEFT require inputs.

It is possible to redefine HEXAGON so that it also requires an input. To do this give the input a name and include it in the title line of the procedure. The name of the input is always preceded by a colon. The input name and the colon are used in the definition of the procedure whenever the value of the input would usually appear. This procedure is understood more easily if you look at an example.

If you call the input for HEXAGON, SIDE, EDIT "HEXAGON puts you in the Edit mode. On the title line, add the input name preceded by a colon so it now reads:

```
TO HEXAGON :SIDE
```

Go down to the next line and delete the 7Ø, and type :SIDE in it's place. The definition of HEXAGON now reads:

```
TO HEXAGON :SIDE
REPEAT 6 [FORWARD :SIDE RIGHT 6Ø]
END
```

If you now type HEXAGON, LOGO will tell you:

```
NOT ENOUGH INPUTS TO HEXAGON
```

HEXAGON is treated like any other LOGO primitive requiring an input. Type HEXAGON 2Ø and the Turtle will draw a hexagon with sides 20 units.

HEXAGON 1ØØ will give a hexagon with sides 100 units, and so on.

PATTERN would have to be rewritten as:

```
TO PATTERN
REPEAT 12 [HEXAGON 7Ø FORWARD 1Ø
RIGHT 3Ø]
END
```

You could modify PATTERN to take SIDE as an input.

```
TO PATTERN :SIDE
REPEAT 12 [HEXAGON :SIDE FORWARD 10
RIGHT 30]
END
```

Use the same principle in CIRCLE to get the Turtle to draw different sized circles.

```
TO CIRCLE :SIZE
REPEAT 72 [FORWARD :SIZE RIGHT 5]
END
```

You are really drawing a 72 sided polygon rather than a true circle, although the result is an acceptably smooth curve. The turtle draws circles rather slowly, but you can speed things up by making the Turtle invisible, then LOGO will not have to draw it at every step. To do this use HIDETURTLE, which can be shortened to HT.

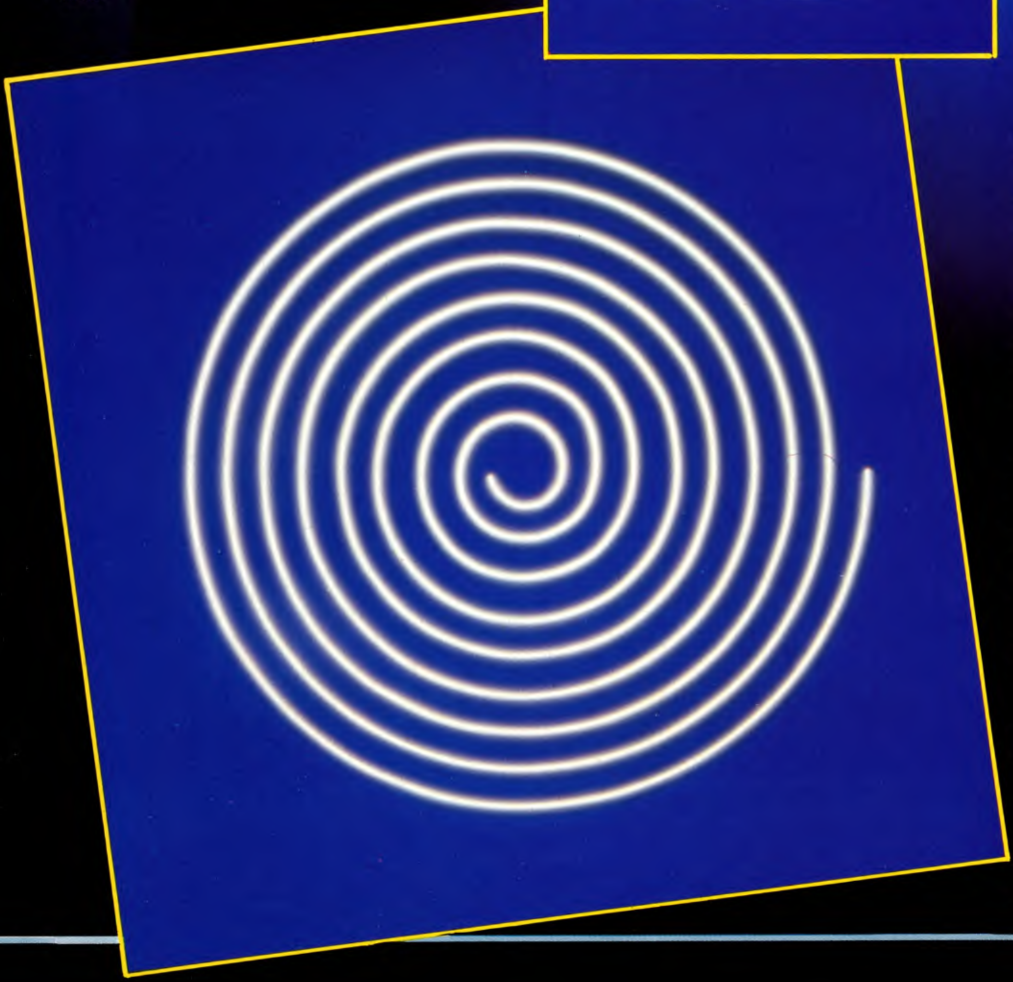
When you want the Turtle visible again use SHOWTURTLE, shortened to ST. Modify CIRCLE to speed it up:

```
TO CIRCLE :SIZE
HIDETURTLE
REPEAT 72 [FORWARD :SIZE RIGHT 5]
SHOWTURTLE
END
```









KEEP IT ON FILE

When a procedure is defined it becomes part of LOGO's vocabulary, but switch the computer off and your own primitives disappear from the memory. LOGO allows you to save your work on disk or cassette by using SAVE followed by ". If you forget the ", LOGO will not save your work and all your efforts will be erased.

When using SAVE, LOGO puts the complete contents of your 'workspace'—every procedure you have defined—onto disk or cassette. SAVE "WORK would save all the procedures in the computer's memory on disk, creating a file called WORK.

After loading, LOGO can retrieve the file from disk or cassette with LOAD (sometimes READ). LOAD "WORK will take all the procedures from the file named WORK and load them into the computer's workspace.

If there is a procedure in the computer's memory and you load another procedure with the same name, the procedure loaded into the memory will replace the one already there, which, like the crew of the Marie Celeste, will be lost without trace.

Files on disk can be erased from the disk with ERASEFILE " and the name of the file. Sometimes ERF is used instead of ERASEFILE. So ERASEFILE "WORK erases all procedures under the filename WORK from the disk. It is not possible to use ERASEFILE with a cassette. To erase a file on a tape you must save another workspace in the same place as the file you want to erase.

Using ERASEFILE does not affect the procedures currently in the computer's memory only procedures already saved.

Because LOGO saves the procedures in the memory on an 'all or nothing' basis, it is sensible to sort out the contents before saving. There are several LOGO primitives concerned with workspace management.

CHECKING MEMORY

LOGO's working memory is a list of nodes. Each node consists of five bytes. After loading LOGO, there are up to 3000 nodes to play with. These are used up as procedures are defined or loaded. To find which procedures are in your computer use POTS. This is Print Out TitleS and prints out the title of every procedure in your workspace. For example:

```
POTS
TO HEXAGON :SIDE
TO PATTERN :SIDE
TO FLOWER
TO CIRCLE :SIZE
```



PO stands for Print Out. PO followed by the name of a procedure prints out the definition of that procedure. For example:

```
PO CIRCLE
TO CIRCLE :SIZE
HIDETURTLE
REPEAT 72 [FORWARD :SIZE RIGHT 5]
SHOWTURTLE
END
```

POALL stands for Print Out ALL. It prints out the definition of every procedure in the workspace. After finding out which procedures are in the workspace you may want to erase some or all of them.

ERALL stands for ERase ALL, and erases all procedures in the memory.

To erase an individual procedure or group of procedures use ERASE. To erase a single procedure, precede the procedure's name with quotation marks. For example: ERASE "FLOWER removes the procedure FLOWER from the workspace.

To erase a group of procedures, enclose their names in square brackets. For example: ERASE [FLOWER CIRCLE PATTERN] erases those procedures from the computer's memory.

CATALOG prints a list of the files stored on the disk that you are using.

When procedures are erased from the workspace the nodes are released ready to be used again. They are not added to LOGO's present list of free nodes. LOGO continues working with it's original list until it is used up. It then searches the memory for nodes which have been freed and forms them into a new list. This is known as garbage collection, and sometimes causes LOGO to pause for a second or two.

ROUND AND ROUND

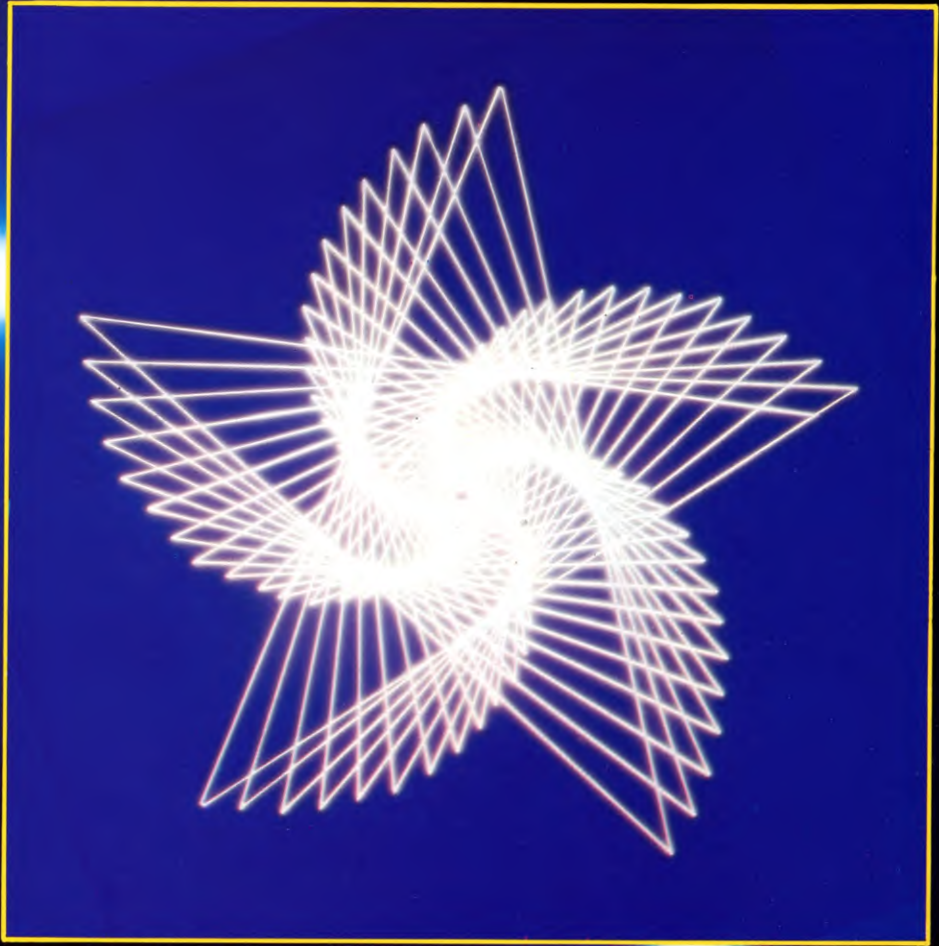
And now back to the drawing board. Here's another way to draw the hexagon.

HEXAGON 50, for example, the Turtle goes FORWARD 50 RIGHT 60 and then must do HEXAGON2 50 again and again and again until you stop it.

Just as one procedure can call another procedure as part of its defintion, it can also call itself as part of itself. This introverted state of affairs is known as recursion (see pages 1289 to 1295).

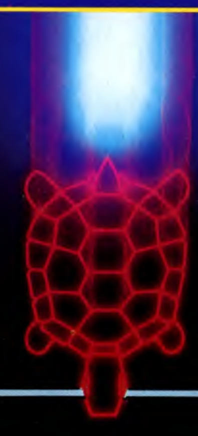
A procedure can take more than one input. Here is a simple example:

```
TO RECTANGLE :SIDE1 :SIDE2
REPEAT 2 [FORWARD :SIDE1 RIGHT 90
FORWARD :SIDE2 RIGHT 90]
END
```



```
HEXAGON2 :SIDE
END
```

If you try and walk this one out you'll get dizzy and collapse. It draws a hexagon of a given size, just as HEXAGON did. The Turtle behaves very differently, however. In HEXAGON the Turtle stopped when it had completed the drawing. In HEXAGON2 the Turtle keeps going round and round until you stop with the BREAK or CTRL key plus G key. The reason this happens is that if you type



RECTANGLE 20 40 will draw a rectangle witn sides 20 and 40 units.

The input can be an angle or a side.

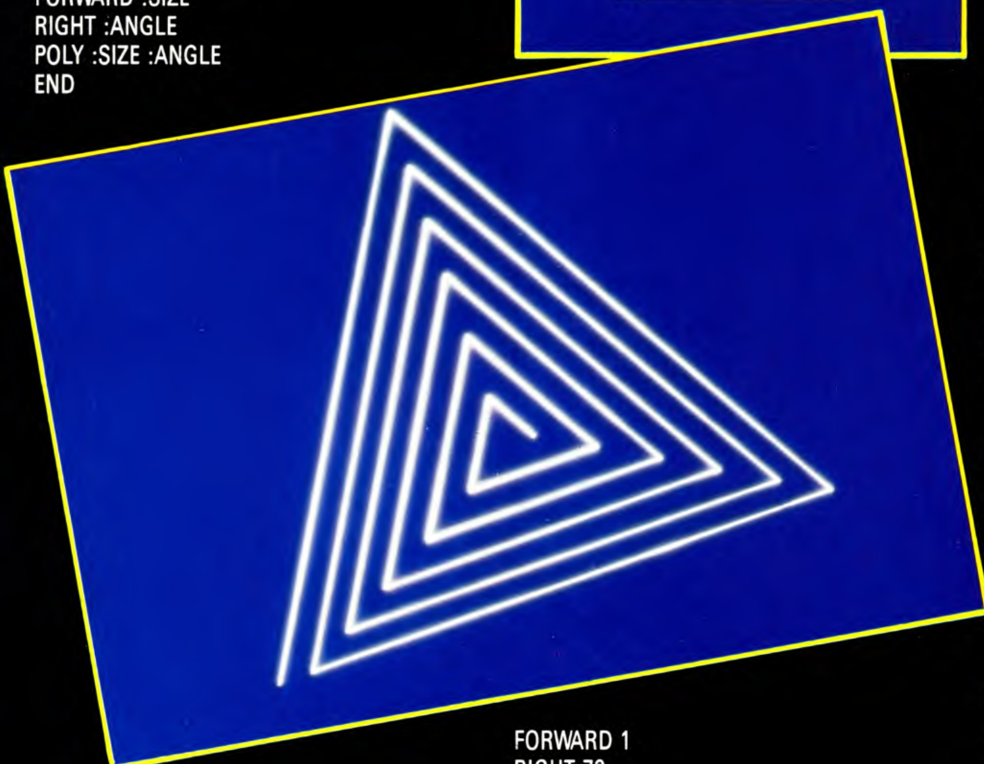
```
TO WIGGLE :SIDE :ANGLE
REPEAT 60 [FORWARD :SIDE RIGHT :ANGLE ]
END
```

WIGGLE takes one input for its FORWARD step and one for the amount of turn.

A common LOGO procedure similar to WIGGLE is POLY.

```
TO POLY :SIZE :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
POLY :SIZE :ANGLE
END
```





POLY is a simple recursive procedure which can produce some attractive designs.

Like HEXAGON2, POLY will set the Turtle a never ending task. You must stop it by interrupting the program.

An interesting development of POLY, which goes beyond simply repeating the same two commands is SPIRAL.

```
TO SPIRAL :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
SPIRAL :SIDE+2 :ANGLE
END
```

LOGO allows full use of mathematical functions, so in the fourth line it adds *2* onto the value of SIDE.

Typing SPIRAL 1 72 will make the Turtle do the following sequence:

```
FORWARD 1
RIGHT 72
FORWARD 3
RIGHT 72
FORWARD 5
RIGHT 72
FORWARD 7
RIGHT 72
FORWARD 9
```

and so on.

This produces a pentagonal spiral. Each side of the spiral is 2 steps longer than the previous side.

FULL STOP

There is no way to stop the Turtle other than by interrupting the program, but you can impose a condition within the procedure to stop the Turtle drawing when the side reaches a certain size.

STOP is a LOGO primitive which can only be used within a procedure. It cannot be used in the immediate mode or at 'top level' where the programmer is in control and LOGO is waiting for an instruction. It stops the procedure before it reaches the end statement and returns control to the 'caller'. The caller can be another LOGO procedure. If there is no other procedure then control goes to the user at top level. STOP only halts the procedure that it appears in.

You can also create a condition with IF and THEN. Some versions do not require THEN. IF looks at something and decides if it is true or not. If it is true THEN a certain course of action is set into motion. If the condition is not true then another course of action is pursued.

Inside a procedure IF tests a condition, if it is not true then the next line of the procedure is executed. If it is true, THEN tells LOGO what to do next.

You can see STOP IF and THEN in action in the following procedure.

Substitute HEXAGON for SIDE in SPIRAL, this will make the Turtle draw a spiral of larger and larger hexagons. The conditionals will stop the procedure when the hexagon's sides reach 20 units.

```
TO SPINHEX :SIDE :ANGLE
IF :SIDE>100 THEN STOP
HEXAGON :SIDE
RIGHT :ANGLE
SPINHEX :SIDE+3 :ANGLE
END
```

LOGO recognizes > meaning 'greater than' just as it recognized the mathematical function of +.

SPINHEX 1 10 tells the Turtle to draw a hexagon with sides 1 unit, turn 10 degrees, draw a hexagon with sides 4 units, turn 10 degrees and draw a hexagon sides 7 units and so on, increasing the size of the hexagon by 3 units each time. Each time the second line checks to see if the size of the sides are greater than 100. If they are not, it passes down to the next line and draws another hexagon. When the size becomes greater than 100, THEN comes into action. It points LOGO to STOP and the procedure stops.

These articles have already covered enough geometric material to last a ten year old for a year. Recursion makes it possible to obtain a wide variety of results from a simple program. The screen editor makes it easy to tinker around with the procedures and encourages experimentation. The results of the programs are often pleasantly surprising. The Turtle translates commands into pictures and makes it easy for you to see where the bugs lie.

In the third and final part of this article, we will look at LOGO's sprites, word and list processing and mathematical capabilities.

CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

COMING IN ISSUE 42...

- ***Running a program takes on a whole new meaning as you attempt to save your valuable carpet from the cascading paint in DESPERATE DECORATOR***

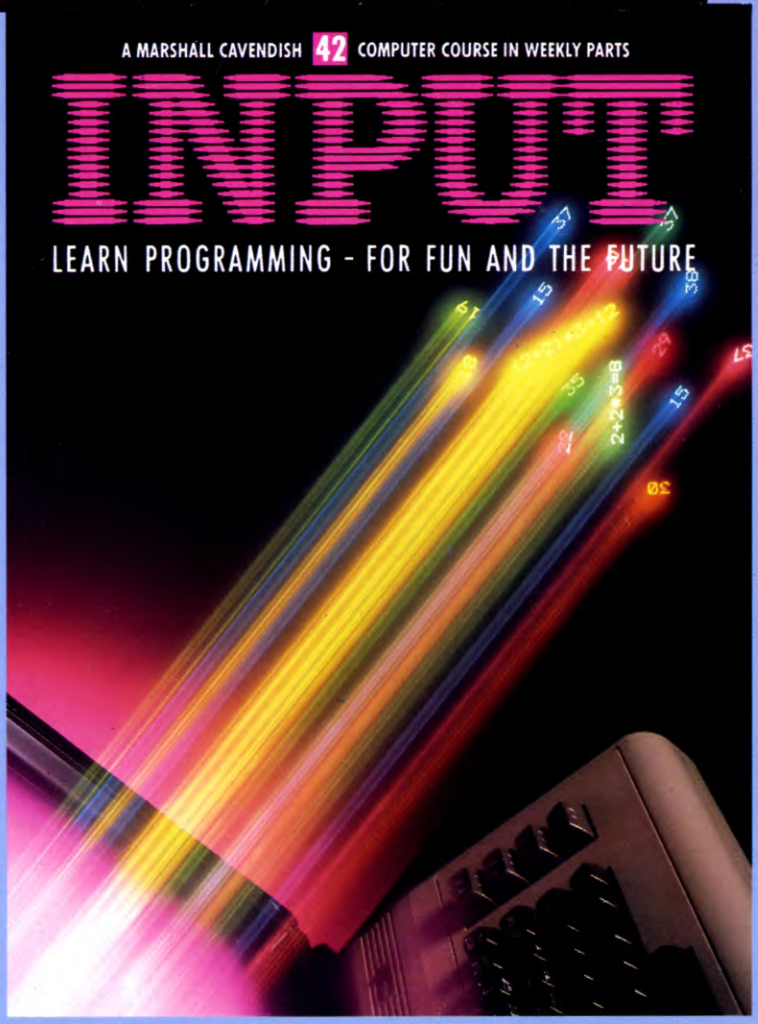
- ***Find just the place for your grand piano, chaise-longue and Chippendale with part two of ROOM DESIGNER***

- ***From your vantage point above the battlefield, issue ORDERS to your troops in WARGAME***

- ***Lights, action, roll 'em! WILLIE will have to be jumpin' jack flash to avoid these rolling stones***

- ***The OPERATING SYSTEM stops your computer being a useless pile of silicon junk. Get to grips with its workings***

- ***Leave the Turtle behind and explore more of LOGO including mathematics, sprites and word handling***







ASK YOUR NEWSAGENT FOR INPUT